AFRL-IF-RS-TR-2005-272
Final Technical Report
July 2005

PASIS: A DISTRIBUTED FRAMEWORK FOR PERPETUALLY AVAILABLE AND SECURE INFORMATION SYSTEMS

Carnegie Mellon University

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. H549/J368

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

STINFO FINAL REPORT

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-2005-272 has been reviewed and is approved for publication

APPROVED: /s/
PATRICK M. HURLEY
Project Engineer

FOR THE DIRECTOR: /s/
WARREN H. DEBANY, JR.
Technical Advisor
Information Grid Division
Information Directorate

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 074-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing this collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	July 2005	**Final** **Jun 99 – Dec 03**

4. TITLE AND SUBTITLE

PASIS: A DISTRIBUTED FRAMEWORK FOR PERPETUALLY AVAILABLE AND SECURE INFORMATION SYSTEMS

5. FUNDING NUMBERS

G - F30602-99-2-0539
PE - 62301E
PR - H549
TA - 10
WU - 01

6. AUTHOR(S)

Gregory R. Ganger
Pradeep K. Khosla

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh PA 15213-3890

8. PERFORMING ORGANIZATION REPORT NUMBER

N/A

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Defense Advanced Research Projects Agency
3701 North Fairfax Drive
Arlington VA 22203-1714

AFRL/IFGA
525 Brooks Road
Rome NY 13441-4505

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

AFRL-IF-RS-TR-2005-272

11. SUPPLEMENTARY NOTES

AFRL Project Engineer: Patrick M. Hurley/IFGA/(315) 330-3624 Patrick.Hurley@rl.af.mil

12a. DISTRIBUTION / AVAILABILITY STATEMENT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

12b. DISTRIBUTION CODE

13. ABSTRACT ***(Maximum 200 Words)***

This document reports on the results of the DARPA project entitled "PASIS: A Distributed Framework for Perpetually Available and Secure Information Systems", funded by Air Force award F30602-99-2-0539. The PASIS project developed architectures and technologies for information storage systems that can survive attacks and failures, both at the small – and the large – scale. Specifically, PASIS storage systems are designed to be:

Perpetually Available: Information should always be available, even when some system components are down or unavailable.

Perpetually Secure: Information integrity and confidentiality policies should always be enforced, even when some system components are compromised.

Graceful in Degradation: Information access functionality and performance should degrade gracefully as system components fail.

Achieving these objectives in practice also requires that performance and usability features of conventional (non-survivability) storage systems be maintained. Therefore, the PASIS project also developed trade-off management, programming, and administrative tools for PASIS systems.

14. SUBJECT TERMS

Survivable storage, self-securing storage, storage security, distributed storage, distributed file systems, fault tolerance, Byzantine, erasure coding, threshold schemes, intrusion tolerance, attack trees, PASIS

15. NUMBER OF PAGES 203

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

LIST OF FIGURES

LIST OF TABLES

1 INTRODUCTION

This document reports on the results of the DARPA project entitled "PASIS: A Distributed Framework for Perpetually Available and Secure Information Systems," funded by Air Force award # F30602-99-2-0539. The PASIS project developed architectures and technologies for information storage systems that can survive attacks and failures, both at the small- and the large-scale. Specifically, PASIS storage systems are designed to be:

- Perpetually Available: information should always be available, even when some system components are down or unavailable.
- Perpetually Secure: information integrity and confidentiality policies should always be enforced, even when some system components are compromised.
- Graceful in degradation: information access functionality and performance should degrade gracefully as system components fail.

Achieving these objectives in practice also requires that performance and usability features of conventional (non-survivable) storage systems be maintained. Therefore, the PASIS project also developed trade-off management, programming, and administrative tools for PASIS systems.

1.1 Approach

PASIS is a survivable storage system, providing for the confidentiality, integrity, and availability of stored data even when some storage nodes fail or are compromised by an intruder. The PASIS architecture builds on decentralized storage systems by encoding and distributing data across many storage nodes. Attackers may compromise some storage nodes, but the rest will continue to protect stored data and provide access to legitimate users.

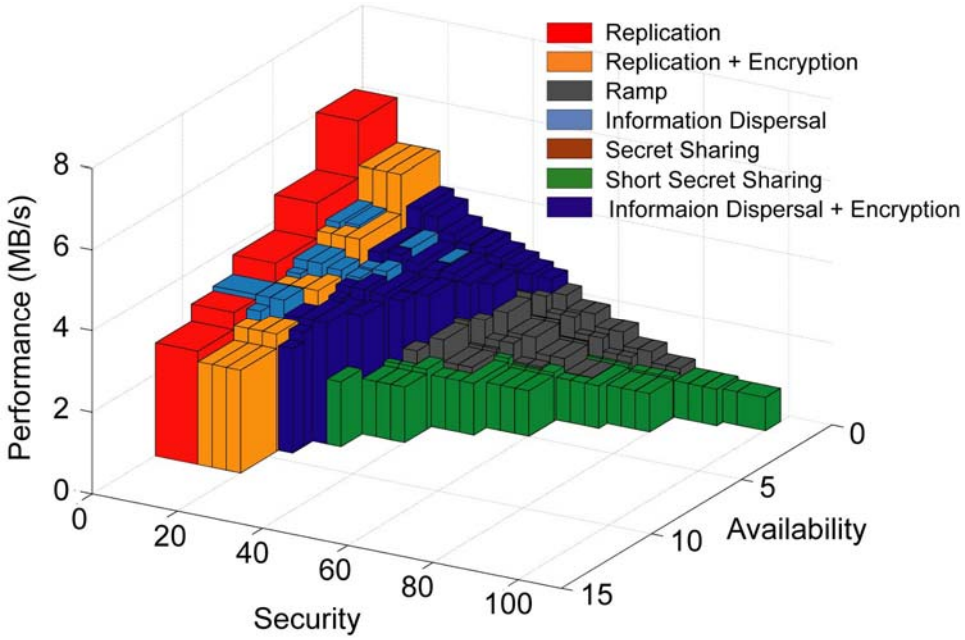


Figure 1-1: Scheme selection trade-offs. Data distribution scheme selection surface plotted in trade-off space. The trade-off space has performance, availability, and security axes. Performance is quantified as the number of 32 KB requests per second that can be satisfied for a single client; only the best-performing scheme that provides at least the given security and availability levels is shown, resulting in a monotonic decrease along those axes. Availability is the probability that stored data can be accessed. These parameters are explained further in Chapter 4.

Because PASIS-style storage can carry substantial performance costs, much of our effort was focused on developing an engineering understanding of these costs and how to mitigate them. There is an inherent trade-off amongst security, availability, and performance in a survivable storage system. We built the PASIS system to allow flexibility in the key design decisions, particularly those related to how data is encoded and distributed among storage nodes. Using this flexibility, we developed models of the performance, security, and availability of stored data. Figure 1-1 illustrates this trade-off space. These models allow engineering analysis to identify which encoding and distribution schemes best meet the overall requirements for a particular storage system. Implementation techniques such as careful server selection and over-requesting can improve client performance even further.

1.2 Contributions

The PASIS project has been quite large, and has resulted in many important contributions and the completion of all specified tasks. Although this report details the full effort, some notable results are highlighted in the following sections.

1.2.1 General architecture for survivable storage

One of the project's first activities was to define a clean architecture for survivable storage systems (detailed in Chapters 2 and 3, and in "Survivable Information Storage Systems" [Wylie00]), which consists of client-side agents dispersing data across multiple servers. By doing so, and fetching multiple fragments during reads, client data is protected from the compromised of a subset of servers and availability is protected from server failures. This architecture has been the centerpiece of the PASIS effort, and also captures the high-level behavior of most point systems from the research literature. Specific systems and configurations are differentiated in the algorithms used to spread data across storage nodes, their parameters, and the communication protocols.

1.2.2 Framework for reasoning about trade-offs

The PASIS project tackled the difficult problem of reasoning about the engineering trade-offs inherent in configuring a PASIS survivable storage system. Each of the thousands of possible configurations yields a different point in a complex trade-off space among performance, availability, and security (confidentiality and integrity). We found that no choice is right for all systems, and we developed an approach to codifying and visualizing this trade-off space, as illustrated in Figure 1-1. Using this approach, we explored the sensitivity of the space to system characteristics, workload, and desired levels of security and availability. Doing so involved creating models of each of these axes. Chapter 4 describes the new availability model with support for tractable exploration of the effects of failure correlation. Chapter 5 describes an approach to quantifying confidentiality. Chapter 6 describes performance models for PASIS systems. The trade-off visualization approach, as detailed in Chapter 7, will also work with most other models.

We used measurements from our initial prototype to configure the default parameters and to validate the performance components of the model. The approach and its use are described in "Selecting the Right Data Distribution Scheme for a Survivable Storage System" [Wylie01], a CMU/SCS technical report available on the PASIS web site (http://www.pdl.cmu.edu/PASIS/).

1.2.3 Final prototype achieves performance and flexibility

The PASIS project involved several phases of implementation. Based on our experiences with a first PASIS prototype, measurements of the system, and explorations of the trade-off space, we designed and implemented a second PASIS prototype. The final system supports much more configuration flexibility and much more efficient storage services. We have demonstrated instances across a wide range of data encoding schemes, numbers of failures, etc. [Wylie03a, Wylie03b, Goodson03b, Goodson03c].

1.2.4 Completely decentralized concurrent updates to PASIS storage

We developed a family of protocols for supporting concurrent updates to PASIS-ized storage in a completely decentralized way (Chapter 8). It builds on storage nodes that keep multiple versions of each data item to scale effectively and avoid the need for inter-server agreement protocols. Each member of the protocol family provides provably correct consistency under a different set of failure and timing model assumptions, ranging from crash/crash/synchronous to Byzantine/Byzantine/asynchronous. Further, the client-server interface and server functionality remain unchanged across all protocols in the family; only the read and write quorum sizes are affected, allowing a single design to be fit to all deployment options. The initial work on this protocol was described in a CMU/SCS technical report, "Decentralized Storage Consistency via Versioning Servers" [Goodson02], and the family is described in a second technical report, "Efficient Consistency for Erasure-coded Data via Versioning Servers" [Goodson03a]. These new protocols have been integrated into the PASIS prototype.

1.2.5 Multiversioning storage node created

Chapter 9 describes the work in which we developed a multiversioning storage node, called the S4 self-securing storage node, to support the decentralized consistency protocols. Comprehensive versioning also allows a storage system to survive client-side and user account intrusions. By versioning all data and auditing all requests within each storage node, security administrators can be given a wealth of information for detecting, diagnosing, and recovering from such intrusions. To make such versioning efficient, we developed novel space management schemes for metadata-intensive workloads and better indexing schemes for traversing the history of versions. These have been implemented in the S4 prototype, and we have found them to be highly effective, doubling the length of history that can be kept. Our measurements indicate that our versioning system performs quite reasonably (better than BSD NFS and less than 15% overhead compared to no versioning). The paper "Self-Securing Storage: Protecting Data in Compromised Systems" [Strunk00] appears in Proc. of the 4th Symposium on Operating Systems Design and Implementation (OSDI), and "Metadata Efficiency in a Comprehensive Versioning File System" [Soules03] was presented at 2nd USENIX Conference on File and Storage Technologies (FAST).

1.2.6 Programming support for working with PASIS storage

We designed an infrastructure to make legacy code PASIS-enabled. This includes the C and Java interfaces to PASIS, which will allow programmers to directly interact with the PASIS libraries. We also experimented with using compiler support to extract an application's requirements for survivability. Such information can be used by a PASIS infrastructure library to adapt the method and parameters of data distribution. A paper describing results in this area, "Testing the Portability of Desktop Applications to a Networked Embedded System," [Bigrigg01] was published and presented by Mike Bigrigg in the Workshop on Reliable Embedded Systems (held with the IEEE Symposium on Reliable Distributed Systems). Information on this subject is also available in Chapter 10.

1.2.7 PASIS file system demonstrated, running Oracle database

We built and demonstrated a functional PASIS file system, which allows PASIS storage to be used for existing COTS applications. To demonstrate it, we ran TPC database benchmarks atop PASIS as a way to characterize the overhead of using PASIS in the JBI context (see Chapter 11). The TPC benchmarks (outlined in Chapter 12) include JBI-like activities (e.g., data mining) and gave us a useful indication of the impact of replacing conventional storage with a survivable storage system. These experiments were a key part of demonstrating that PASIS could play a useful role in hardening JBI systems with acceptable performance overheads.

1.2.8 Decentralized reconfiguration protocols developed and evaluated

We developed and evaluated mechanisms for decentralized reconfiguration of data distributions in survivable storage systems. Such reconfiguration is crucial for long-term survival, and decentralized recon-

figuration avoids creating a new central point of failure or compromise. Our focus was on devising protocols that allow verifiable, decentralized redistribution; that is, the servers should be able to collectively determine that the final shares comprise the original data object without ever actually recombining the object. Two CMU/SCS technical reports, "Verifiable Secret Redistribution" [Wong01] and "Verifiable Secret Redistribution for Threshold Sharing Schemes" [Wong02a] originally presented this material. Advances to this work, described in the paper, "Verifiable Secret Redistribution for Archive Systems" [Wong02b] were presented at the First IEEE Security in Storage Workshop.

1.3 Technology transfer

The PASIS project included a validation effort (like many OASIS projects; see Appendix A) and substantial technology transition efforts along two fronts: military and industry.

Working with AFRL JBI researchers, we explored the costs and benefits associated with using PASIS technologies in Joint Battlespace Infosphere (JBI) systems. This involved analyzing available JBI literature, meeting with the JBI people several times including a JBI PI meeting, and exploring different ways that PASIS could fit in. We concluded that the most natural fit was to replace base JBI storage with PASIS-enhanced storage. To experiment with the consequences of doing so, we ported the Oracle database atop our PASIS prototype. Our demonstration of and results from running the TPC-H benchmark atop Oracle on PASIS-ized storage showed its feasibility. This JBI activity represented an important (and early) first step towards transfer of PASIS technologies to critical military applications.

To help further AFRL understanding and DEM/VAL efforts, we shared copies of the prototype software with other DARPA/AFRL supported organizations. We have tried to help them understand the nature of the prototype, which was designed for experimentation rather than deployment.

The PASIS team has also spent large amounts of time explaining the PASIS technologies to industry leaders from the dozen companies (including IBM, HP, Sun, Intel, EMC, Veritas, Microsoft, and Oracle) who support their research. Over the years, several weeks worth of interactions have refined the practicality of PASIS and its acceptance by the various companies. The most important of these meetings have been the annual PDL Retreat and PDL Open House events, during which 20-30 technical leaders from these member companies hear about and discuss our research activities. Based on these interactions, we expect to see some aspects of the PASIS concepts appearing in products, available to DoD users, in the near term. This is the most effective way for technology transfer of infrastructure research projects, such as PASIS, to achieve DoD deployment.

1.4 PASIS Document Overview

The rest of the PASIS document is structured as follows: Chapter 2 presents a high-level overview of PASIS technology. Chapter 3 reviews the implementation of the PASIS prototype system, focusing on high-level functionality. Chapters 4, 5 and 6 describe the trade-off framework of PASIS, looking at PASIS availability, security and performance in detail. Chapter 7 explores the trade-offs associated with PASIS-enhanced storage. Chapter 8 describes a family of consistency protocols that exploit data versioning within storage-nodes to efficiently provide strong consistency for erasure-coded data. The first half of Chapter 9 describes self-securing storage and our implementation of a self-securing storage server, called S4 as well as the versioning components of self-securing storage. The second half of the chapter describes and evaluates two methods of storing metadata versions more compactly: journal-based metadata and multiversion b-trees. Chapter 10 explores exception handling and the design and implementation of four software exception injection (SWEI) systems and the use of exception analysis with survivable storage alternatives. Chapter 11 describes the Joint Battlespace Infosphere {JBI) environment and outlines how PASIS technologies can fit into and enhance the survivability of JBI systems. Chapter 12 reports on experimental database benchmarking results.

2 AN OVERVIEW OF PASIS

PASIS is, first and foremost, a storage system. It therefore must provide the application semantics of standard storage such as *read*, *write*, and *create*. From the rest of the system's perspective, interfacing with PASIS is essentially equivalent to interfacing with other file systems[1].

The primary goal of PASIS is to enhance the survivability of storage systems. By survivability, we mean the continuing provision and the graceful degradation of the system performance, availability, and security, under fairly generous assumptions such as malicious storage servers. Survivable storage systems such as PASIS differ from reliable storage systems. Reliable storage systems that aim at fail-stop faults [Siewiorek92] while tolerating Byzantine faults are a fundamental requirement for survivable storage systems.

To meet its survivability goals, PASIS employs the fundamental design theory that no individual service, storage node, or user can be fully trusted; having compromised entities in the system must be viewed as the common case rather than the exception. Therefore, data, as well as services, must be replicated and distributed across multiple nodes and platforms [Wylie00].

The PASIS architecture combines three fundamental technologies to achieve survivable information storage,

- *Decentralized storage systems*: A decentralized storage system eliminates single failure points by entrusting the data's persistence to sets of nodes rather than individual nodes, and therefore is an ideal candidate platform for building survivable storage
- *Threshold data encoding*: By varying the threshold size, PASIS provides a framework for users to trade off performance, information availability and confidentiality.
- *Dynamic self-management*: Truly survivable systems require the ability to perform data maintenance automatically and instantaneously.

Availability and confidentiality of data are the two primary goals of storage systems. Many existing systems handle these goals independently—straightforward replication for availability and encryption for confidentiality. In such systems, it is often not easy to trade-off one goal against the other. Threshold schemes offer an alternative in which a piece of information is encoded, replicated and divided into a set of shares, of which only a subset is needed to reconstruct the information. These shares are then distributed across different storage nodes. A threshold scheme provides data confidentiality because any unauthorized set of shares yields little or no information (an adversary must compromise a large enough set to reconstruct the original data), and availability because only a subset of the storage nodes needs to be available to supply information. By varying the number of shares required to reconstruct the original data, one can vary and trade off the level of system availability and confidentiality.

Threshold data-encoding schemes are best explained by the jigsaw puzzle metaphor. It is analogous to creating a "1000 piece puzzle" out of the information and dispersing these puzzle pieces to different computers. For example, by examining 20 pieces of a jigsaw puzzle shown in Figure 2-1, one cannot extract the map information because many other pieces of the puzzle are still missing. Further, if each piece of the puzzle is small enough then gaining access to any piece does not allow access to information because each piece is "cognitively encrypted," i.e., it does not possess "coherent information."

In the PASIS system, client-side agents communicate with the collection of storage nodes to read and write information, hiding the decentralization from the client system. Automated monitoring and repair agents on storage nodes provide self-maintenance features.

[1] PASIS does allow users to supply additional parameters to specify their storage schemes.

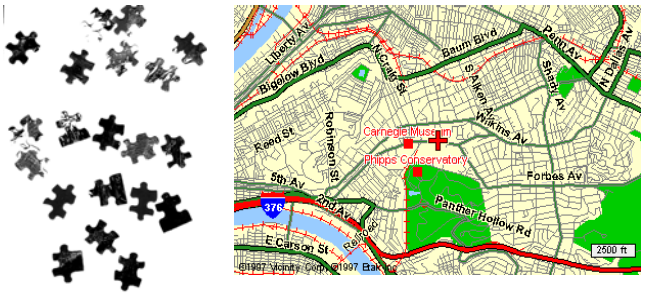


Figure 2-1. Puzzle pieces and completed puzzle.

Several current and prior efforts employ similar technologies to achieve survivability. For example, the Intermemory project [Goldberg98] and the Eternity Service proposal [AndersonR96] use decentralization and threshold schemes for long-term availability and integrity of archived (write-once) digital information. The e-Vault [Iyengar98] and Delta-4 [Deswarte91] projects do the same for on-line (read/write) information storage; Delta-4 additionally addressed information confidentiality and other system services (e.g., authentication). These projects have helped advance the understanding of the technologies and their roles in survivable storage systems, but such systems have not yet achieved widespread use.

The rest of this chapter investigates the core technologies that comprise survivable information storage systems and discusses their impact on the key characteristics of the PASIS design. Information in this chapter has been previously published in "Survivable Information Storage Systems" [Wylie00], "Selecting the Right Data Distribution Scheme for a Survivable Storage System" [Wylie01] and "Survivable Storage Systems" [Ganger01].

2.1 Decentralized Storage Systems

Most existing distributed storage systems rely on, at most, a few servers to store information. These servers are targeted failure points for accidents and malicious attacks. Replacing centralized approaches with highly decentralized systems is a first step towards survivable information storage. Prior work in cluster storage systems, such as xFS [AndersonT96], NASD [Gibson98], and Petal [Lee96], provides much insight into how to construct decentralized storage services efficiently while maintaining a single, unified view to applications. Our design draws heavily from the experiences of the cluster storage research as well as other distributed storage systems work [Bolosky00b].

In decentralized systems, information is partitioned among nodes using data distribution and redundancy schemes commonly associated with disk array systems (e.g., RAID), ensuring scalable performance and fault-tolerance. For example, the Petal system uses chained declustering, in which data blocks are striped across storage nodes, with each data block replicated on two nodes. When users need to access data, they contact a subset of the storage nodes in the system to collect the desired blocks of information. Although decentralized storage systems were designed primarily for the purpose of scalability and fault tolerance, their elimination of single failure points provides a starting point for survivable storage systems.

2.2 Data Redundancy and Encoding

PASIS uses threshold schemes for data redundancy and encoding. Threshold schemes, also referred to as secret sharing schemes, divide a piece of data into n shares in such a way that any set of m or more shares ($m < n$) can reconstruct the information, but sets of up to (m-1) shares do not reveal any information about the original data. This type of threshold scheme is referred to as an (m, n) scheme.

General threshold schemes have been studied extensively in the field of cryptography, and various schemes have been proposed. For illustration purposes, we describe two threshold schemes—Blakley's [Blakley79] and Shamir's [Shamir79]. However, the PASIS system is not tied with any particular threshold schemes.

Blakley's threshold scheme works in an *m* dimensional space where the data to be shared are points in the space and shares are multidimensional planes. Consider the example of implementing a (2, 3) threshold scheme for secret *v* (i.e., the secret is divided into three shares and the knowledge of at least two shares can reconstruct *v*). Blakley's scheme works as follows. First, a random point *A* is generated in a two-dimensional space so that the *y*-coordinate of *A* is *v* (see Figure 2-2). We then generate three random lines that intersect at *A* and distribute the three polynomials, which determine the lines, to three computers. It is clear that knowledge of at least two lines enables the calculation of the intersection point, *A*, from which *v* can be easily extracted. However, a single computer, knowing at most one polynomial, cannot derive any information about *v* because *A* can be any point on the line.

Another common threshold scheme is Shamir's algorithm [Shamir79], which is based on interpolating the coefficients of polynomials by using results of the polynomial at certain points. The original secret is encoded as the zero term coefficient in the polynomial, and the various shares are the result of evaluating

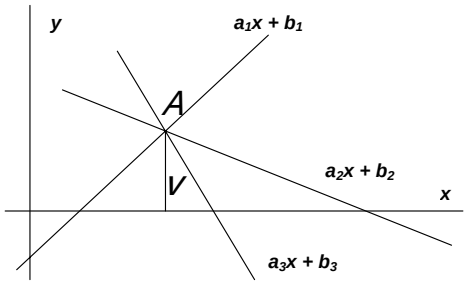


Figure 2-2. Simplified Blakley's Secret Sharing Scheme

the polynomial for different values. If the number of result values known for a polynomial (i.e., shares) is larger than the degree of the polynomial, then it is possible to calculate the coefficients of the polynomial through interpolation, thus extracting the original secret. It is mathematically infeasible to calculate the coefficients of a polynomial if the number of result values is less than or equal to the degree of the polynomial.

The topic of robust threshold secret sharing in the presence of malicious shareholders has also been studied. Verifiable Secret Sharing (VSS) [Feldman87] and Proactive Secret Sharing (PSS) schemes [Herzberg95] are designed for exactly this purpose. An interesting approach is the verification and redistribution of shares without reconstructing the original data [Wong02]. Because the data never appears at any one place in its entirety, even during redistribution between one set of share holders to another, these secret sharing schemes can render a high degree of data confidentiality.

Note that a threshold scheme can be information-theoretically secure; that is, even an adversary with *infinite* computing power cannot learn anything about the secret without first gaining access to a large enough set of shares (i.e., larger or equal to the threshold). The adversary can do no better than randomly guess the secret value.

Also note that we can generalize threshold schemes to include simple replication, decimation, splitting and information dispersal schemes. These schemes are usually not information-theoretically secure because each share may leak useful information. To capture the amount of information leak, we introduce a third parameter *p.* A (*p*-*m*-*n*) scheme divides the data into *n* shares, and any *m* or more shares can reconstruct the secret. A group of fewer than *p* shares gain no useful information about the secret. For example, a scheme that simply replicates the data into *n* copies/shares is a (1, 1, *n*) scheme because each share reveals useful information and reveals the information in its entirety. Several threshold schemes that the reader may be familiar with by different names are depicted in Figure 2-3.

Replication (*1-1-n*)
Replication increases information availability by increasing the storage required by a factor of *n*. Replication provides no information confidentiality, since each "share" contains an entire copy of the original information object.

Splitting (*n-n-n*)
Splitting increases information confidentiality by increasing the storage required by a factor of n. Splitting decreases information availability since all *n* shares must be available. Splitting can be implemented with simple operations, such as eXclusive-OR.

Decimation (*1-n-n*)
Decimation divides information objects into *n* pieces and stores each piece separately. Decimation decreases information availability since all shares must be available; it offers no information-theoretic confidentiality, since each share exposes 1/*n* of the original information.

Information Dispersal Algorithm (*1-m-n*)
Rabin's Information Dispersal Algorithm offers a range of information storage solutions that trade off between information availability and required storage. Information dispersal algorithms can be thought of as a combination of the replication and decimation techniques. Like decimation, they do not offer confidentiality.

Figure 2-3. Examples of Simple Threshold Schemes [Rabin89]

To meet availability and confidentiality concerns, existing information systems use straightforward replication and client side encryption. For survivable information storage systems, threshold schemes offer distinct advantages over replication and encryption. First, threshold schemes provide a great deal of versatility, because information may be divided into a variable number of shares of which a variable fraction can be required to reconstruct the information. This flexibility creates a trade-off between information confidentiality and availability. For example, fewer shares required (i.e., a lower *m* value) means that fewer storage nodes must survive attacks to maintain availability, but also that fewer storage nodes must be compromised for an attacker to learn information. Secondly, most options within the spectrum of trade-offs are more space-efficient than maintaining replicas on each node. Third, encoding and decoding in some threshold schemes are orders of magnitude faster than public key cryptography. Finally, threshold schemes provide information confidentiality from storage nodes without requiring users or client systems to remember (or share) secret keys.

2.3 Dynamic Self-Maintenance

Over time, all systems need maintenance. The decentralized nature of survivable storage systems makes maintenance difficult and therefore error-prone. Truly survivable systems should automatically perform at least some self-maintenance in order to avoid undesirable conditions. Self-maintenance includes regular monitoring to identify potential problems, such as failed or compromised nodes, performance bottlenecks, and denial of service attacks. For example, failure of a storage node reduces the number of additional failures that can occur before information is lost. As with recovering from a disk failure in a disk array, full

redundancy can be restored by rebuilding the contents of the lost storage node onto a spare replacement. Another example of such self-maintenance involves adjusting threshold scheme parameters based on observed performance and reliability.

2.4 The Trade-Off Space of PASIS

To compare PASIS to conventional information storage systems, consider a PASIS installation with fifteen storage nodes that uses a (*3-3-6*) threshold scheme versus a conventional installation. In the PASIS system being considered, shares are uniformly distributed among storage nodes. The conventional system uses fifteen servers organized into five server groups. Each server group consists of a primary server and two back up servers. Thus, each server group is responsible for 20% of the information stored. Figure 2-4 summarizes the results of this comparison.

		PASIS	Conventional
Confidentiality	Percentage of information revealed if one storage node is compromised.	0%	20%
	Percentage of information revealed if three storage nodes are compromised.	4.4%	Up to 60%
Availability	Probability that system cannot serve a read request, given that each node fails with probability 0.001.	1.5×10^{-11}	10^{-9}
Performance	Reading small objects.	Significantly slower performance for PASIS	
	Reading large objects.	Similar	

Figure 2-4. Comparison of PASIS and Conventional Systems

Confidentiality is a measure of a system's ability to ensure that only authorized clients are privy to stored information. To provide confidentiality of information, complete copies of the original information should only exist on client nodes with an authenticated, authorized user. Conventional systems store complete copies of information on storage nodes — this in itself is a failure to provide confidentiality. To breach a conventional system's confidentiality, an intruder needs to compromise only a single storage node that has a replica of the desired object. On the other hand, if one storage node is compromised in a PASIS system that uses secret sharing, no confidentiality is lost. Even if m=3 storage nodes are compromised, the confidentiality of only 4.4% of the information is compromised. In a conventional system, the confidentiality of up to 60% of the information stored in the system is lost when three storage nodes are compromised.

Availability is a measure of a system's ability to serve a specific request. For PASIS to serve a specific read request, at least m of the n storage nodes that have a share must be available. Thus, three of the six storage nodes that contain shares of the data must be available in the PASIS system to serve a request. In a conventional system, as long as one of the servers in the desired server group is available, a read request can be served. These numbers make it appear as though the conventional system has better availability than the PASIS system. However, the real measure of a system's availability is the probability that it can-

not serve a specific read request. For comparison purposes, assume that there is a 0.001 probability of each storage node not being available at a given time (i.e., storage nodes fail independently with probability 0.001). The probability that the PASIS system cannot serve the read request is 1.5×10^{-11}, while the probability that the conventional system cannot serve the read request is 10^{-9}. In this comparison, the PASIS system has two orders of magnitude better availability than the conventional system.

Performance in the form of latency is a measure of the delay experienced when a system serves a request. In a conventional system, a read or write request is served by exchanging messages with a single server. In a PASIS system, messages must be exchanged with multiple storage nodes, which can significantly impact performance. However, the situation is not as bad as might be expected. For example, the information dispersal algorithm and short secret sharing schemes require that m servers each provide S/m of a dispersed object's S bytes. For large objects, network and client bandwidth limitations can potentially hide the overhead of contacting m servers. Still, for small objects, PASIS systems may require up to an m-fold increase in the number of small messages, whose processing can dominate performance. Also, all m replies must be received before a request is complete, meaning that the slowest server dictates the response time.

Distributed storage schemes can be fine-tuned for performance, availability and security. For example, a threshold scheme divides data into n shares such that any m of the shares can reconstruct the information and fewer than p shares reveal no information. By varying the value of m, we can achieve a different level of confidentiality, availability and system performance. Other algorithms, such as information dispersal, decimation and replication, provide thousands of data distribution schemes for survivable storage systems. Thoughtfully selecting one scheme from this set requires the ability to model the effect of the scheme on the various system properties. Having accurate models for availability, security, and performance is therefore essential to storage system design. This section describes our approach to comparing the performance, availability, and security of various schemes.

Evaluating Availability: The substantial body of prior work in building highly-available systems [Gray91, Siewiorek92] provides a clear metric for evaluating information availability: the probability that a desired piece of information can be accessed at any given point in time. Assuming uncorrelated failures, this probability can be computed from the probabilities that required system components are available. For example, a *p-m-n* threshold scheme requires at least m of the n storage nodes containing shares to be operational to perform a read. If f_{node} is the probability that a storage node has failed or is otherwise unavailable, then the availability of the stored information is

$$Availability_{read} = \sum_{i=0}^{n-m} \binom{n}{i} (f_{node})^{i} (1 - f_{node})^{n-i} \qquad (2.1)$$

For writes, the computation of availability depends upon the system's design. A system could require that all of n specific nodes be operational for a write to succeed. This approach clearly has poor availability characteristics. To improve write availability in a system with $N>n$ storage nodes, the write operation can attempt to write shares to different storage nodes until it has completed n writes. Alternatively, a system could allow a write to complete when fewer than n shares have been written. This requires more work during failure recovery or reduces read availability of the stored data (because n is effectively lower for that data). We calculate write availability using the last approach. Thus, m storage nodes of the N storage nodes in the system must be available for a write to be performed, and write availability is

$$Availability_{write} = \sum_{i=0}^{N-m} \binom{N}{i} (f_{node})^{i} (1 - f_{node})^{N-i} \qquad (2.2)$$

Availability requirements for storage systems tend to be quite high. A popular manner for discussing high-availability values is in terms of "nines," referring to the number of nines after the decimal point in the availability value before a digit other than nine. For example, a low availability value of 0.993 has just

two nines, whereas a high availability value of 0.99999996 has seven nines. It is worth noting that failures are not always uncorrelated, as is generally assumed in availability computations. For survivable systems in particular, denial-of-service (DoS) attacks can induce highly correlated failures, bringing into question the value of this metric. We believe that the relative availability levels computed for different schemes provide insight even with DoS attacks. Although the absolute values may not always be meaningful, relative values certainly are when comparing data distribution schemes for a given set of system components. In particular, availability increases only when more options are available for servicing requests. Thus, a higher availability value means that a DoS attack must eliminate a larger set of storage nodes.

Availability modeling with correlated failures is a much more complex problem. We investigate this issue further in Chapter 4.

Evaluating Security: By far, security is the most difficult of the three dimensions to evaluate, as there are no proven metrics or even a mature body of research from which to draw. Our initial plan was to reuse the mathematics of fault tolerance, counting how many storage nodes must be compromised to bypass confidentiality or integrity. However, this approach raised significant problems: First and foremost, it relies upon having a "probability of being compromised" with which to compute security values. Unfortunately, we are aware of no reliable way to obtain or even estimate such a value. Further, using such a value would be questionable, because security problems experienced by nodes in a distributed system are expected to be highly correlated. When the system is under attack, this probability value goes up. Another problem with evaluating security in terms of a probability of a node being compromised is that it is a useful measure for only a subset of the data distribution algorithms (threshold algorithms). For example, it ignores the additional confidentiality provided by encryption.

Our current approach to evaluating security focuses on the *effort, E,* required for an active foe to compromise the security of the system. For example, for an *n-n-n* threshold scheme, breaking into all *n* heterogeneous storage nodes or can compromise confidentiality by compromising the authentication system:

$$E_{Conf} = \min\left[E_{Auth}, \left(n \times E_{BreakIn}\right)\right] \tag{2.3}$$

Extending this example, assume that the data is also encrypted and that the names of shares reveal no information about the encoded data. The attacker has many paths to the data — attempted cryptanalysis, theft of the encryption key, attempted combination of shares in many permutations or compromising the directory service. Assuming that an attacker is going to take the easiest path to the data:

$$\begin{aligned} EConf = & \min\left[E_{Auth,}\left(n \times E_{BreakIn}\right)\right] \\ & + \min\left[E_{Cryptanalysis,}\, E_{StealKey}\right] \\ & + \min\left[E_{IdentifyShares,}\, E_{StealNames}\right] \end{aligned} \tag{2.4}$$

In this paper, we focus strictly on the security of the storage system; attacks on the authentication service and directory service are not considered further. Thus, we use only two values in our security model: $E_{BreakIn}$ and $E_{CircumventCryptography}$. These values are dimensionless. Thus, the unit of the security axis is "effort units," and security values across all runs have been normalized on a scale that ranges from 0 to 100.

The first term, $E_{BreakIn}$, is the effort required to steal a piece of data from a storage node. The second term, $E_{CircumventCryptography}$, is the effort required to circumvent cryptographic confidentiality. It is a coarse metric that includes cryptanalysis, key guessing, and the theft of decryption keys. The confidentiality of an encrypted replica is thus the summation of these two terms: one encrypted replica must be stolen and then the encryption must be broken. Alternatively, the confidentiality of secret sharing is solely a function of the first term (i.e., $m \times E_{BreakIn}$) — a total of *m* shares must be stolen to compromise the confidentiality of data, thus the effort is *m* times the effort to steal a piece of data. We calculate the effort to compromise the confidentiality of information dispersal as a weighted sum of the information content of the shares, where the weight is proportional to the number of shares that must be stolen to decode. This heuristic sets the

confidentiality of information dispersal above that of replication, below that of secret sharing, and higher as m increases. We assume that $E_{BreakIn}$ is constant for all storage nodes (i.e., distinct attacks of equivalent difficulty are necessary to compromise each storage node). For this assumption to be true storage nodes must be heterogeneous. Beyond this, we hypothesize that, as n increases, the likelihood that an attacker can find a storage node that they can compromise increases; as such, confidentiality should decrease as n increases. We do not currently consider n in the calculation of security. A more detailed security model can be implemented, however these simple models capture the major features of the algorithms we are currently investigating.

Security is often defined as availability, confidentiality, and integrity. In our analysis, we have distinguished availability from the two other security characteristics because there is a trade-off between it and them. We focus our security axis on confidentiality, since there appears to be agreement in the community that cryptographic hashes are the right way to protect integrity, and we have found no contradictory evidence. The hashes can be generated from the cleartext or the ciphertext, and they can be stored with data shares, encoded into the name, or stored in the directory service. All of these increase integrity significantly; usually well beyond other effort levels.

Although effort terms are difficult to quantify, we believe that effort-based evaluation focuses the designer's attention on the right thing — raising the security bar "high enough." As with availability, the relative merits of schemes can be compared usefully even if the absolute effort quantities are inaccurate. Further, other security engineering research projects are now addressing the problem of measuring security and doing so in terms of effort [Ortalo99]. An evolving security model is presented in Section 4. As better effort models become available, they can be modularly inserted into the trade-off exploration approach.

Evaluating Performance: As with availability, performance metrics and evaluation techniques of distributed systems are part of a mature body of prior work [Jain91, Kleinrock73]. The main issue faced in creating a general tool is balancing detail, which should yield greater accuracy, with generality in the performance models. We use a simple, abstract system model to predict the relative performance of different data distribution options. We intend for this model to represent a wide array of survivable storage system designs. The model includes three parts: CPU time for encode or decode in the intermediary software, network bandwidth for delivering the shares, and storage node response times.

CPU Time. Encode and decode operations involved with any data distribution scheme require CPU time. There are orders of magnitude differences between the CPU times for different schemes. For example, Figure 2-5 shows the CPU cost for encoding a 32 KB block for four *p-m-n* threshold schemes for all possible parameter selections with $n \leq 25$. Notice the large differences between the shapes of the performance curves and the times at the same m and n for different schemes.

We have constructed and calibrated simple models for the following data distribution algorithms: ramp schemes, secret sharing, information dispersal, replication, decimation, splitting, short secret sharing, encryption, and hash algorithms. The models require CPU measurements of a few key primitives: polynomial interpolation of order $m-1$, random number generation, triple-DES encryption, and MD5 hash generation. Given the requisite measurements, the models can predict the CPU time required for an encode or decode operation for any of the data distribution schemes with at least 90% accuracy.

Network Bandwidth. The amount of data that must pass between the client and the storage nodes depends directly on the data distribution scheme. For any of the *p-m-n* threshold schemes, the network bandwidth required depends on the read-write ratio and the values of p, m, and n. Each share's size can be computed as the original data size multiplied by $1/(m-(p-1))$. For a write, we assume that all n shares must be updated. For a read, we assume that only m shares are fetched by default.

We model the network bandwidth with a distribution, indicating the probability of a given bandwidth during any period of time, assuming that a single bottleneck link determines the aggregate bandwidth. Al-

though imprecise, we believe that this allows the first-order effects of concern to be captured, when combined with the storage node latencies discussed below. For a local-area network or dial-up client, we expect this bottleneck link to be at the client's network interface card. For a wide-area system, we expect this link to be at the edge router through which the client interacts with the wide-area network. Network congestion would appear in this very simple model as a reduction of available bandwidth (if consistent) or an increase in variability (if sporadic).

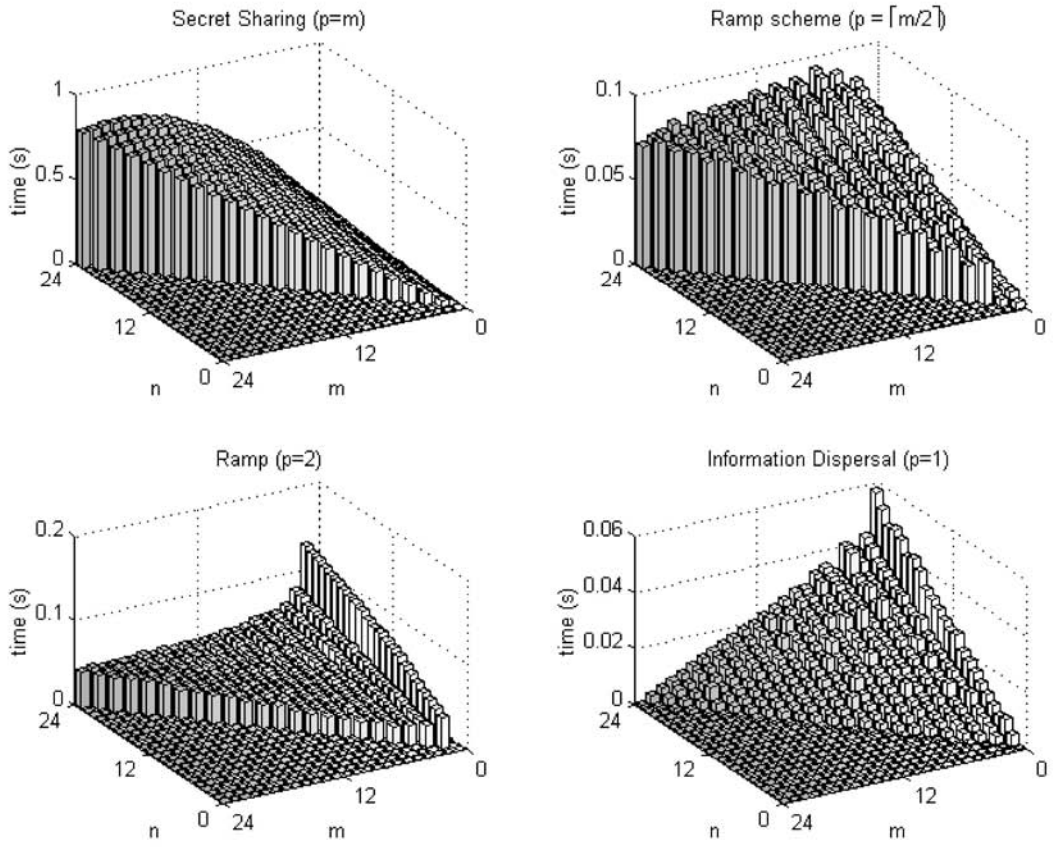


Figure 2-5. Measured encode time for 32 KB blocks on a 600 MHz Pentium III

Storage Node Latency. A read or write request to a storage node involves work at the storage node in addition to moving data over the network. This is observed at the client as a response time latency that includes both delays, but accurate modeling requires separating these delays and recombining them appropriately. In particular, the time for an operation that simultaneously writes data to *n* nodes must capture both the shared bandwidth and concurrent latency aspects.

We model storage node latency as a random variable whose mean and variance depend on the operation type (read or write) and the request size. Dependence on the former is expected, and we found dependence on the latter necessary to model storage protocols (e.g., FTP or NT4's CIFS implementation) that opens a new TCP/IP connection with slow-start for each file transferred. Competing loads on a storage node would appear in this simple model as increases in the mean and/or the variance.

Overall performance model. To predict the access latency for a given request, we combine these partial models as follows. For a write, ignoring variance and assuming homogeneous servers, we sum the modeled CPU time for encode, the storage node latency for writing one share to one server, and n times the network transmission time required per share. Thus, in sequence, the data is encoded, each share is sent over the network to its appropriate storage node, and all responses are awaited; the n^{th} request is sent after the first $n-1$ and it finishes after the average storage node latency. For a read, the same steps occur in reverse. With variance, the last request sent is not necessarily the last to finish, so a predicted latency for each request must be computed. Our tools use simulation to compute the expected performance for any given scheme. In our experience, the simulation is fast enough for our purposes, with each single-scheme prediction requiring only 10–30ms. A detail discussion of the performance model can be found in Chapter 6.

3 PASIS IMPLEMENTATION

This chapter describes the implementation of PASIS. The descriptions here focus on high-level functionality in order to give readers a general picture on which the discussions in the later sections are based. More detailed information has been published in "Byzantine-tolerant Erasure-coded Storage" [Goodson03a], "Survivable Storage Systems" [Ganger01] and "Survivable Information Storage Systems" [Wylie00].

For PASIS, it is important to differentiate between the concepts of *blocks* vs. *shares*. A block is a file system construct, while a share is a PASIS construct. A block can be composed of many shares. In a traditional file system, data for the same block appear together in storage. In PASIS, however, shares from the same block are not necessarily stored on the same storage nodes.

From a storage client's perspective, a PASIS-enhanced file system behaves in essentially the same manner as a traditional file system such as NFS or AFS. Figure 3-1 shows the PASIS file system architecture. Our current implementation supports NFS clients.

In Figure 3-1, the client application is a program that initiates file access commands, such as "read" and "write". An example of such a client application is a database server, as in the case of the PASIS implementation. The client program sends a file access command to its local file system interface. In this case the local file system interface is a stub program that simply issues an NFS loopback call. Instead of calling a real NFS file server, the loopback call is directed to the PASIS file system as shown in Figure 3-1. Meanwhile, the client program can input its storage scheme selection (e.g., requirements for performance, security, availability) to PASIS, which in turn creates a storage scheme on behalf of the client.

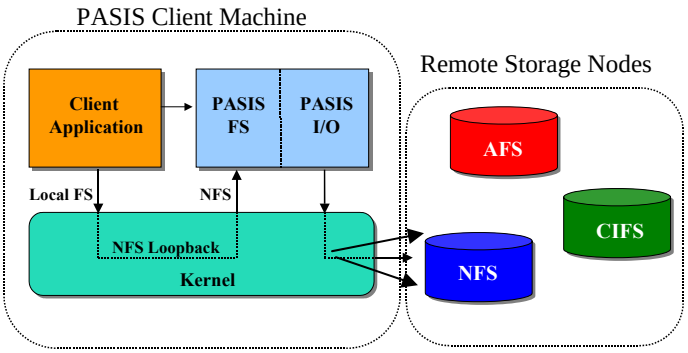


Figure 3-1. The PASIS Architecture

Each data block in PASIS is stored as shares at various remote storage nodes. To find and retrieve shares, a metadata object is associated with the data block. The metadata object describes the locations of each share associated with the block.

Shown in Figures 3-2 and 3-3, PASIS software is divided into two components: PASIS FS and PASIS I/O. PASIS FS translates between client FS calls and PASIS storage requests, and it interacts with the remote storage nodes via PASIS I/O. PASIS I/O selects the set of servers to store the data shares, and create the corresponding metadata object for the block. PASIS I/O is also responsible for preparing shares in a manner consistent with the file scheme. For instance, if the scheme states that the file should be encrypted, PASIS I/O performs encryption on the file blocks and outputs encrypted shares to be stored on the remote storage nodes. For reads, PASIS I/O retrieves shares and reconstruct them into file system blocks before giving the blocks back to the PASIS FS. In the sections below, PASIS FS and PASIS I/O are described in further detail.

3.1 PASIS FS

This section describes the implementation of the PASIS FS (see Figure 3-2). The primary purpose of the PASIS File System (PASIS FS) is to provide the standard file system interface, such as *create, read* and *write,* to client applications, and to interface with PASIS I/O to store and retrieve data. The PASIS FS performs local file caching to improve performance.

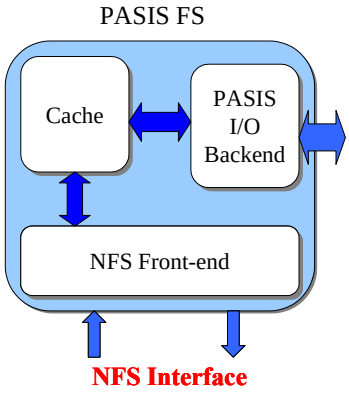


Figure 3-2. PASIS FS

To create a file, PASIS FS generates a scheme object based on user specifications and creates a new inode. It then issues a *create* command to PASIS I/O and passes it the scheme object. As a result of *create,* PASIS FS receives a new fileID and the metadata objects associated with the blocks of the new file. PASIS FS is then responsible for updating the file metadata with the fileID and stores the metadata objects in the inode of the file.

For a *read* request, PASIS FS first searches in its local cache for the necessary data. If the data is present, PASIS FS simply returns the data to the client program. If the data is not found in the cache, PASIS FS requests the data from PASIS I/O, passing it the metadata object that describes the block. Once the data is fetched by PASIS I/O, it is locally cached and passed back to the client program.

For a *write* operation, again PASIS FS searches for the block in its local cache first and updates it if found in the cache (writing a block in the cache would cause the dirty bit to be set). If the block is not in the cache, PASIS FS passes the block metadata object to PASIS I/O and requests the blocks. Once the block is fetched, it is updated and written into the cache.

When a block is accessed, all the metadata objects of its parent blocks must remain in the cache. Periodically, PASIS FS synchronizes its cache with the data on the storage nodes. Only then are the parent metadata objects of a block evicted from the cache. *Create* and *Write-new* are the only operations that will cause a change in the metadata object. The other operations, *read* and *write*, simply use the information in the already defined block metadata.

3.2 PASIS I/O

PASIS I/O handles block-to-share and share-to-block transformations. PASIS I/O takes input commands, such as read, write, or create, from PASIS FS, and interacts with the backend storage nodes to store and retrieve shares. Figure 3-3 depicts the main components of PASIS I/O. This section describes the functionality of PASIS I/O and its implementation.

The primary functions of PASIS I/O are handling *create*, *read*, and *write* requests from PASIS FS. We examine each of the file system requests in detail.

Create: This command creates a new PASIS file.

For a “create” command, PASIS FS passes the following parameters to PASIS I/O:

- UID
- Hint metadata
- A pointer to the new metadata
- A pointer to the new file handle object
- The first block of the file (actual data)

The UID is the unique ID that the file system uses to identify the owner of the file. In an NFS system, the UID is an NFS identifier.

The hint metadata object contains a list of servers that PASIS I/O may use to store the shares of the new file. This hint metadata is produced from a previous write or create operation. The reason that PASIS FS provides hints for file creation is that related files could be placed on to the same set of servers to optimize future retrievals. PASIS I/O, however, can ignore the hints and choose to place the shares onto a

different set of servers should the suggested servers happen to be unavailable. If the hint metadata is not provided, PASIS I/O simply chooses a set of servers from its own list.

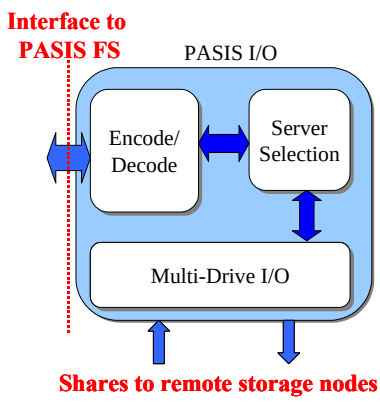


Figure 3-3. PASIS I/O

Internally, PASIS I/O calls the Encode/Decode (E/D) module, passing the data block and the hint metadata. E/D first allocates some temporary space on the client to hold the shares. It then contacts the server selection module to obtain a list of servers on which to store the shares. Next, E/D performs the corresponding encoding operation (replication, encryption, secret sharing, etc.) on the data block. The result of the encoding operation is a list of shares within the allocated share space. As a result, E/D returns either a success or failure status. E/D also generates the metadata for each share.

Once E/D completes its encoding operation, PASIS I/O calls MDIO and passes the share list. Using the appropriate protocol, MDIO talks to the respective storage nodes and writes the designated blocks onto the servers. As a result, MDIO generates a new block metadata object. This object is returned to PASIS FS via call-by-reference.

Write (new): A write-new command is one for which the blocks of the newly created file (with the exception of the first block) get written to the storage nodes. A write-new call from PASIS FS includes the following parameters:

- UID
- Hint metadata
- Pointer to the new metadata (to be generated)
- File handle
- Pointer to the data block to be written

As before, UID identifies the owner of the file. The hint metadata provides suggestions as to a set of servers PASIS I/O may use to store the shares of the block. If PASIS I/O does use the list of servers in the hint metadata, the new metadata is created simply as a copy of the hint metadata. Otherwise, the new metadata contains a list of servers to which the shares of the block are written.

The write-new operation is very much like the create operation except the write-new call does not generate new file handle objects. Rather, it passes in the file handle object generated by the create operation.

PASIS I/O performs a similar set of steps for write-new as for create.

Write (overwrite): An overwrite operation replaces an existing data block without selecting new servers or generating new metadata.

The list of parameters for an overwrite call is similar to that of the create operation. They are:

- UID
- Block metadata
- File handle
- Pointer to the new data block

For overwrite, the hint metadata contains the metadata of the target block. Note that the overwrite call does not pass a pointer for new metadata to be created as a result of the call (as in create). This is because

the overwrite replaces an existing block. It does not generate new metadata or change the location of the shares. Therefore the metadata information is not changed.

For an overwrite, PASIS I/O passes the hint metadata and the data block to E/D. E/D performs its encode operation, allocates temporary space to hold the shares generated from the block, fills the space with the resulting shares, and generates an array of share metadata. Upon the completion of the encoding operation, PASIS I/O passes the pointer to the shares and the share metadata to the MDIO. MDIO then writes the corresponding shares onto the servers indicated in the share metadata.

An overwrite operation is similar to file creation except the server-selection module is not involved. In the current implementation, server selection is performed only at initial file creation. It is likely that a different set of servers will need to be selected when share redistribution becomes necessary.

The return value for the overwrite operation is either success or failure.

Read: To read a block, PASIS FS calls PASIS I/O with the following parameters.

- UID
- Block metadata
- File handle
- Pointer to an allocated space to hold the block

PASIS I/O first calls E/D, passing the metadata, to initialize the appropriate amount of space for the shares and select shares to read. PASIS I/O then calls MDIO to perform the “read the shares” function, passing the metadata. MDIO reads the shares from its respective servers, fills the share space, and returns the metadata object. PASIS I/O then calls the decode method of E/D, passing the pointer to the list of shares. The decode function performs its share-to-block transformation, fills the block pointer with the resulting data, and returns either a success or failure status.

The read operation does not change the block metadata information.

MDIO is the component within PASIS I/O that is responsible for interfacing with the backend storage servers. MDIO contains a job queue, a controller component and a list of I/O threads.

MDIO takes a share list as input. A share list contains pointers to each share within a particular block. Each share is a construct that contains the following fields:

- Operation type (e.g., read, write)
- Data buffer (contains the actual data for the share)
- Server (information about the server where the share lives, name, type, etc.)
- Status
- Pointer to its parent (the share list)
- Path information (PASIS specific path information—where to locate the share on the server)

When MDIO is invoked with a share list, it places the share list into its job queue. The job queue is served in a FIFO fashion. The controller component of MDIO takes the front most share list from the job queue and gives each of its shares to an I/O thread. Each I/O thread performs the designated operation (e.g., write or read) with the designated server, and subsequently fills the status with either success or failure.

Once all the I/O threads have completed, the controller releases the semaphore and wakes up the PASIS I/O thread. PASIS I/O then inspects the status information of each share and if satisfactory, returns control to PASIS FS.

4 AVAILABILITY IN PASIS

This section examines modeling the availability of data distribution schemes. We explain the weakness of the classic approach model, and describe two models that overcome the inaccuracy caused by the independent failures assumption. Our key contribution is a pair of models that preserve the simplicity of the classic model and produce more accurate results than the classic model. We show the efficacy of the model based on conditional probabilities as a design decision tool. We analyze the effects of availability and correlation on the ordering of the schemes. Finally, we investigate the placement of related files. Published work that has contributed to the contents of this chapter includes "On Correlated Failures in Survivable Storage Systems" [Bakkaloglu02] and "Correlated Failures in Survivable Storage Systems" [Bakkaloglu02a].

Survivable storage systems [Wylie00] encode and distribute data over multiple storage nodes to survive failures and malicious attacks. PASIS is one such system that is configurable to support many data distribution schemes, including threshold schemes [Shamir79], for improving storage availability and security. There are many other projects, such as Farsite [Bolosky00] and OceanStore [Kubiatowicz00], which use similar techniques to achieve similar goals. A common property of these techniques is that they have parameters that affect the properties of the system. Setting these parameters in an ad-hoc fashion may be sub-optimal in terms of meeting users' requirements. In [Wylie01], we describe a framework to select these parameters (i.e., a scheme) in the context of performance, security, and availability trade-offs. In general, there are two ways of quantifying these three properties: measurements and models. Models can be parameterized, allowing one to explore a wide range of systems quickly. In this thesis, the focus is on the availability model.

The classic approach to modeling the availability of distributed storage is to assume that storage nodes have identical availabilities and independent failures. This model is useful because it is simple and it enables a system designer to reason about the effect of average storage node availability on system availability. However, it ignores correlations known to exist among storage node failures [Amir96].

The most accurate way of modeling correlated failures is to specify exactly the probability of each subset of nodes being unavailable. But, such a model is too complex in practice and provides little intuition to system designers. A better approach would be to extend the classic model with a single correlation parameter that indicates the level of failure correlation in the system. In this thesis, two such models are described. The first one is based on conditional probabilities, and the second one is based on the Beta-Binomial distribution. Both of these models have two parameters that describe the system: *average storage node availability* and *correlation level*.

An interesting issue in selecting a scheme, for a given system, is the ordering of the schemes based on their availabilities. It may be the case that the user specifies the properties of the underlying system incorrectly. In such a case, if the ordering of the schemes changes, then the scheme selected will be incorrect. We investigate how the ordering of the schemes is affected with respect to average storage node availability and correlation level.

The problem of optimal file placement in a survivable storage system has been studied before [Douceur01b]. In those studies, it is generally assumed that files are independent from one another. In reality, there could be dependencies between files and directories. For example, to access a file, the directories on the path to the file have to be traversed. If the directories are not available, the file will not be available. Thus, the relative placement of related files and directories may impact the availability of the overall system. In this thesis, we analyze cases to show where it is important to consider related files and directories in doing placement.

4.1 Data Distribution Schemes

This section describes data distribution schemes from the literature that are used in survivable storage projects.

Threshold Schemes. Threshold schemes are a way of encoding data. A threshold scheme has three parameters: n, m and p (where $n \geq m \geq p > 0$). Data is divided into n shares, of which any m is sufficient to fully recover the data. Less than p shares give absolutely no information about the data. The formation of the shares is called encoding and the process of combining the shares to reconstruct the original data is called decoding. An example scheme is replication. Each replica is the original data, thus, m=1 and p=1. Other threshold schemes include Information Dispersal [Rabin89], Secret Sharing [Shamir79] and Ramp Schemes [Blakley85]. Table 4-1 lists a number of well-known threshold schemes.

n-m-p	**Name**
n-1-1	Replication
n-n-1	Decimation (Striping)
n-n-n	Splitting (XOR-ing)
n-m-1	Information Dispersal
n-m-m	Secret Sharing
n-m-p	Ramp Schemes

Table 4-1. Threshold Schemes.

In general, the n shares formed during encoding are stored on n different storage nodes. For a file to be available, at least m of the nodes have to be available. For most of the schemes, storing the n shares on different nodes increases availability. This is because the system can tolerate up to (n-m) node failures. Additionally, for an adversary to steal the data in its entirety, she must compromise at least m storage nodes. Security can be further enhanced, by storing shares on nodes with different operating systems. Thus, a weakness in one operating system may not lead to the compromise of nodes with different operating systems. PASIS [Wylie00] and Farsite [Bolosky00] are two example storage systems that use threshold schemes.

Erasure Codes. The idea in erasure codes is similar to threshold schemes in the sense that the data is augmented with redundant information in such a way that any k subset of the n fragments created is sufficient to reconstruct the original data. Erasure codes such as Read-Solomon Codes [Plank97] and Tornado Codes [Luby98] are used in OceanStore [Kubiatowicz00]. Also in [Goldberg98] erasure codes are used.

Quorum Systems. In some data replication protocols, quorum systems are employed. A quorum system is a set of subsets of U (the universe of servers), of which every pair of sets intersect. A read/write operation is done by first selecting a quorum and then by accessing all of the elements in that quorum. Since every pair of quorums intersects, the user always has a consistent view of the system [Amir98]. Optimal quorums for masking b Byzantine failures, where the intersections contain at least $2b+1$ servers are described in [Malkhi00].

The common idea in these different data distribution schemes is that they divide a piece of data into a number of shares and only a subset of the shares is required to fully reconstruct the original data.

4.2 Selecting the Correct Scheme

Parameter options for threshold schemes (i.e., values of n, m and p) create a large space of possibilities. Each scheme has different properties, thus for a given set of requirements the optimal scheme for differ-

ent systems may be different. In order to select the optimal scheme, we first have to be able to compare the schemes and to compare the schemes we need to be able to quantify them. In quantifying the schemes, three metrics are used: performance, confidentiality and availability. Here, we give an overview of each metric. The details of the model for each one can be found in [Wylie01].

Performance. Performance is calculated in terms of throughput (MB per second). The time to produce the shares (i.e., encode the data), the number of shares, the size of each share and the number of shares required to reconstruct the data depend on a schemes' parameters *n*, *m* and *p*. In [Wylie01], we use a simple model to compare the performance of different schemes. Specifically, we examine the effects of changing the workload, the CPU speed at the client side, the network bandwidth and the storage node response times.

Confidentiality. Each scheme offers a different level of confidentiality. This is due to the three parameters *n*, *m* and *p*. As *m* increases, the attacker has to steal more shares to fully reconstruct the file and less than *p* shares give absolutely no information to the attacker. So, higher *m* and higher *p* mean higher confidentiality. On the other hand, having more shares (i.e. higher *n*) degrades confidentiality since there will be more vulnerable nodes. The unit used for confidentiality is the effort required by the adversary to gain information.

Availability. Availability is defined as the probability that a file can be accessed at any given time. With threshold schemes, files are encoded into *n* shares, of which *m* or more are sufficient to fully reconstruct the file. In [Wylie01], we assume the failures of storage nodes are independent and we use the classic availability model [Siewiorek92]. For reads the model is,

$$f(n, m, Avg.Avail) = \sum_{i=m}^{n} \binom{n}{i} (Avg.Avail)^{i} (1 - Avg.Avail)^{n-i} \qquad (4.1)$$

Note that for writes, the availability can be defined in several different ways. One extreme case is to require any *m* nodes to be available and the other extreme case is to require *n* specific nodes to be available. In the former case, read availability gets affected. The unit of availability is nines (i.e. 0.99 means 2 nines).

4.2.1 The 3D Trade-off Space

The section explains how scheme selection is done. We focus on seven classes of schemes: replication, replication with cryptography, ramp schemes, information dispersal algorithms, secret sharing, short secret sharing and splitting. Each scheme is quantified in terms of the three metrics that are explained above. In Figure 4-1, each scheme is represented with a different tone of gray. The graph shows the best performing scheme and its performance for a given confidentiality and availability level. Note that the possible schemes cover only a portion of the entire trade-off space. The reason is that there exists no scheme that can provide maximum confidentiality, maximum availability and maximum performance at the same time. Thus, a user must make trade-offs among the three properties. In order to select a scheme using the 3D Trade-off space, the user specifies a confidentiality and availability level in accordance with her requirements and then uses the graph to identify the optimal scheme in terms of performance.

Default Configuration. The default configuration we use in [Wylie01] consists of 10 storage nodes, 100Mb/sec network bandwidth, 32KB files (blocks), a 0.5 read/write ratio, 0.995 storage node availability and the effort required to circumvent cryptography (e.g., by stealing the key) is equal to the effort required to break into a storage node. Figure 4-1, shows the resulting graph with respect to the seven schemes.

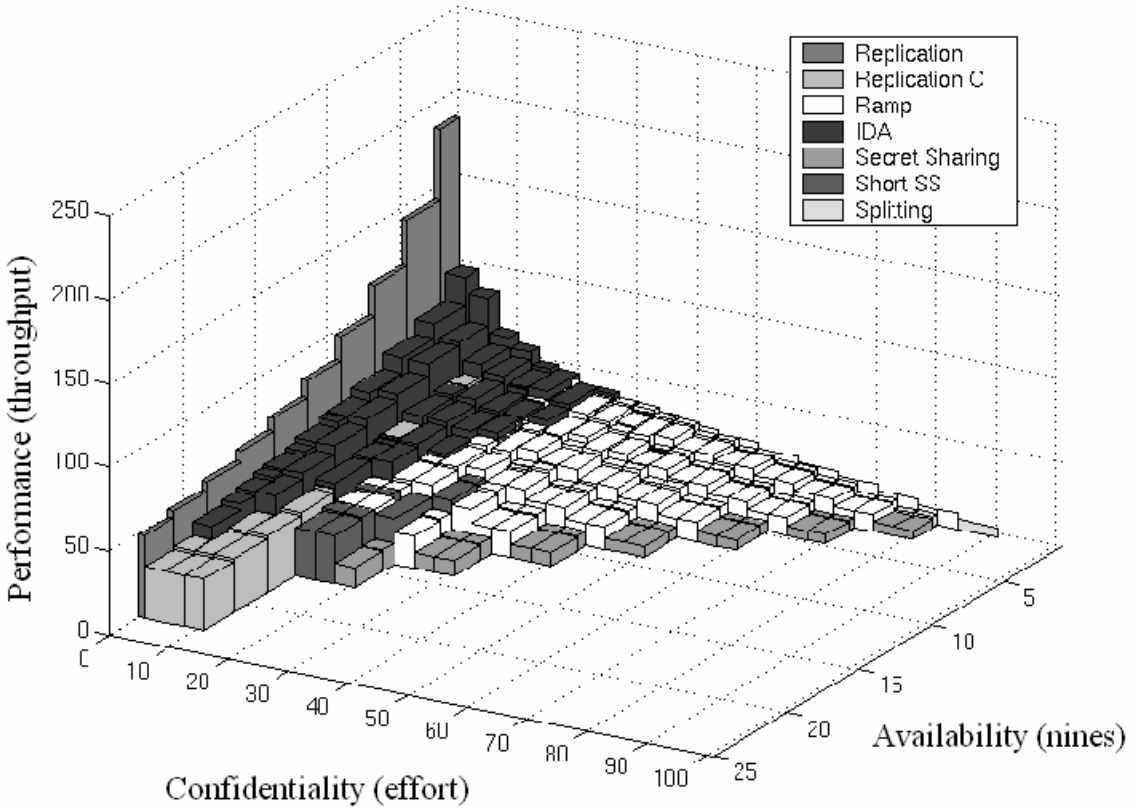


Figure 4-1. Default Configuration

Different Configurations. In analyzing the effects of different system configurations we use the default configuration as the base case. In [Wylie01], we examine the effects of different system configurations such as higher network bandwidth, different read/write ratios in the workload, effort of cryptanalysis being greater than effort required to break into a storage node, and lower storage node availability. The detailed description of the first three is beyond the scope of this paper. Figure 4-2 shows the effect of having lower storage node availability.

An inaccurate estimate of average storage node availability can drastically affect the space that is covered by the schemes. For example, if the storage node availability is estimated as 0.95, as opposed to 0.995, the original graph (i.e., Figure 4-1) gets contracted and covers a smaller portion of the space. This causes the user to become unaware of the fact that there are schemes that provide higher levels of availability. Furthermore, if the user selects a scheme using the contracted graph, the selected scheme would have a lower performance than the scheme that would have been selected, if the correct graph had been used. These imply that the optimal scheme, for a given set of requirements, heavily depends on individual storage node availabilities. Also an interesting phenomenon is the change in the ordering of the schemes as average availability changes. A detailed analysis of the ordering of the schemes is given in Chapter 5.

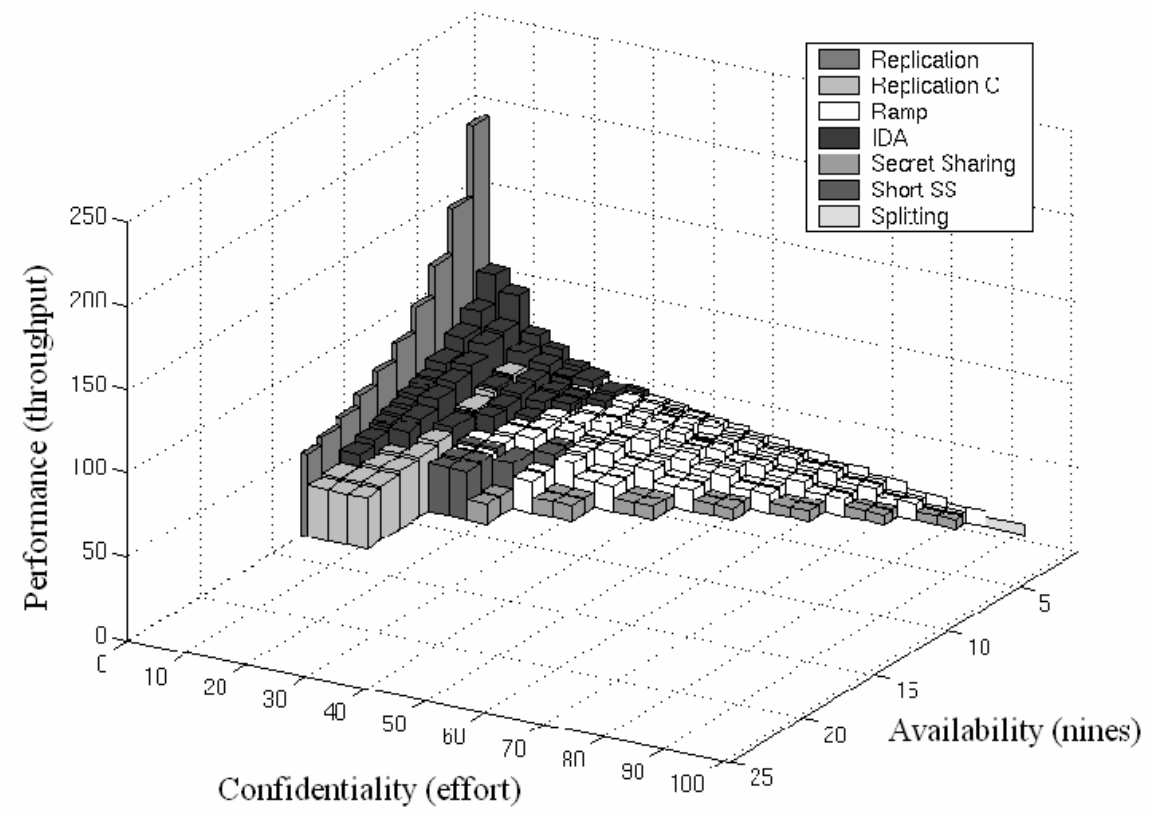


Figure 4-2. Lower Storage Node Availability

4.2.2 Discussion

As discussed in the previous section, using simple models for performance, confidentiality and availability enables us to explore the trade-off space among the three properties. Different configuration parameters result in different selection surfaces. Another important point is that each model has its inherent assumptions that affect the 3D graph. For example, in the availability model it is assumed that storage node failures are independent. In the next section, we analyze the availability model in more depth.

4.3 Modeling the Availability of Threshold Schemes

In this section, we investigate modeling the availability of threshold schemes. The ideas here can be applied to modeling the availability of other data distribution schemes. First, we describe the basics and analyze the underlying assumptions of the classic way of modeling availability. In Section 4.3, we explain two approaches to correlation-aware availability modeling and we compare the two models to real measurements.

In modeling the availability of threshold schemes, we treat the individual storage nodes as black boxes in the sense that we only know when the node is up or down and we have no knowledge of why this has happened. We assume that only the client link failures are detectable and these are the only failures that are not counted as the server being unavailable. We model the availability of a scheme using the availability values of the black boxes and we do not have a clear picture of the relations between black boxes. More specifically, we have no knowledge of the underlying network and the different administrative domains.

4.3.1 Availability of Threshold Schemes

The availability of a storage node is defined to be the percentage of time the node is able to service requests. One common way of expressing it is as nines of availability (e.g., 0.99 is 2 nines) [Gray91]. There are two kinds of availabilities for threshold schemes: read and write availability. A file is available to being read if there are *m* or more up-to-date shares that are available at the time of read. On the other hand, there are several ways that write availability could be defined,

- Any *n* of the *N* storage nodes in the system being available (e.g., creating a file, rewriting a file)
- Any *m* of the *N* storage nodes in the system being available (e.g., creating a file, rewriting a file)
- All *n* of *n* specific nodes being available (e.g., updating a file)
- Any *m* of *n* specific nodes being available (e.g., updating a file)

Clearly, the way writes are done affects the availability of reads. In our model, we focus on read availability; it is straightforward to extend our model to write availability.

4.3.2 The Classic Model

In the literature [Siewiorek92], a common way of modeling the availability of a given set of storage nodes is by assuming:

- The failures of the storage nodes are independent and
- The storage nodes have identical availabilities that are equal to the average availability.

The formula used in this case is:

$$f(n,m,Avg.Avail) \quad = \quad \sum_{i=m}^{n} \binom{n}{i} \quad (Avg.Avail)^{i} \quad (1-Avg.Avail)^{n-i} \qquad (4.2)$$

Independence Assumption. To see how the independence assumption of the classic model affects the outcome, consider the following example shown in Figure 4-3. Assume two machines each with availability of 0.99. The classic model's estimation for n=2 and m=1 is 0.9999 (4 nines). But, in reality, the availability of this scheme could be anywhere between 0.99 and 1.0. If the downtimes of the two nodes completely overlap, then replication provides no benefit. On the other hand if the downtimes do not overlap at all then the availability of the scheme is 1.00.

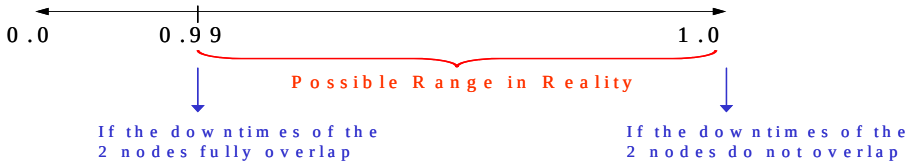


Figure 4-3. Availability of scheme with n=2 and m=1

In the following sections, we show that, by relaxing the independence assumption, the accuracy of the model can be improved substantially.

4.3.3 Correlation-Aware Availability Modeling

4.3.3.1 Requirements

The classic model shown in the previous section is an easy tool to use. It is a function of the form,

$$f(n, m, Avg.\ Avail.) \tag{4.3}$$

As explained in Section 4.2 the independence assumption of the classic model may lead to inaccurate estimations. The definition of independence is,

$$P(\text{Node X down} \mid \text{States of other nodes in the system}) = P(\text{Node X down}) \tag{4.4}$$

The independence assumption is not always valid. Several studies show that storage node failures are correlated. In Bolosky [2000] and Douceur [2001c], the actual scheme availabilities (replication in their case) differ from the calculations that assume independent failures. In [Amir96], they analyze 14 machines in a wide area network and report that crashes are correlated. Our measurements of web server availabilities also support the fact that,

$$P(\text{Node X down} \mid \text{Node Y down}) > P(\text{Node X down}) \tag{4.5}$$

Therefore, the new model should have the following properties:

- It should be an easy to use tool, which means that it should not have too many parameters.
- It should be more accurate than the classic model for cases when there are correlations between failures.
- While achieving the first two it should not hurt cases when the failures are actually independent.

In achieving these requirements, we investigate the possibility of relaxing the independent failures assumption without overly complicating the model. More specifically, the required function is of the form,

$$f(Avg.\ Avail.,\ n,\ m,\ Correlation\ Level) \tag{4.6}$$

In the following sections, we describe our datasets and two approaches to correlation-aware availability modeling. The first approach is based on conditional probabilities. The second approach is based on the Beta-Binomial distribution. The Beta-Binomial distribution has been used previously to model correlated events such as the number of infectious diseases in a household [Griffiths73] and failures in multi-version software [Nicola90].

4.3.3.2 Datasets Used

In our work we use two datasets. The first one is our availability measurements of Web Servers and the second one is measurements of Desktops [Bolosky00].

Web Servers. This dataset consists of our measurements of more than 100 web servers from September 2001 to December 2001. The web servers come from various domains such as com, edu and gov, located in various countries. The client was an Intel 600MHz Linux machine. The data was collected, by reading a file off each web server every 10 minutes. If the server responded and if the file was valid (e.g., not a HTTP 404 Error and has HTTP OK in its header), then we recorded this as the server being available. In this sense, the availability measured is the service availability. In our experiments, we exclude some of the servers due to reasons such as protocol version incompatibilities. We use 99 of the servers (they are listed in the Appendix B-1). The failures between the web servers were found to be correlated. In order to have more cases to compare our models against, in addition to using the entire dataset, we also use subsets of the entire dataset. In this thesis, we show results from 4 representative datasets. Dataset-1, Dataset-3 and Dataset-4 are subsets taken by time and Dataset-2 is the entire dataset. Dataset-1 is weeks [1-3], Dataset-3 is weeks [8-10] and Dataset-4 is week 8.

Desktops. This dataset consists of anonymized desktop availability statistics measured at Microsoft Corporation [Bolosky00]. Machines, in the campus, were pinged hourly from July 1999 to August 1999. The failures between the machines in this set were almost independent. Here, again to have more cases to compare our models against, we analyzed subsets of the entire dataset. In this thesis, we show results from 4 representative subsets where each one is constructed using a subset of the machines. Dataset-1 is machines [1-2000], Dataset-2 is machines [1001-3000], Dataset-3 is machines [2001-4000] and Dataset-4 is machines [4001-6000].

4.3.3.3 The Model Based on Conditional Probabilities

Calculating Correlation

The most comprehensive way of determining correlation is to calculate the probability of each subset of nodes being unavailable. But, such a model is a difficult tool to reason about the effects of correlation on the availability of different schemes. A simpler approach would be to use a single correlation parameter in the model. Therefore, the question is "How can such a parameter be measured?" According to the classical definition of correlation, two variables have a correlation close to 1, if they mostly take on the same value and they have a correlation close to –1, if they mostly take on opposite values. The problem with using this approach for our case is that, if two machines are up 99% of the time, they will have a high correlation value regardless of how correlated their failures are. Therefore, this way of calculating correlation is not suitable for nodes with high availabilities.

A better approach is to calculate the average conditional probability. A similar approach was used in modeling the correlated failures in multi-computer systems [Tang92]. Average conditional probability is calculated as follows.

$\forall$ Pair of nodes (X, Y) in the system, where $X \neq Y$,

Calculate P (Node X is down | Node Y is down)

The average of all these probabilities is the correlation level of the system.

[Note that for two nodes A and B, P (Node A down | Node B down) is not necessarily equal to P (Node B down | Node A down).]

When the correlation level is 0, this means that there is absolutely no overlap between the down times of storage nodes (note that 0 does not mean independence). When it is 1, it means that all of the down times are overlapping.

Another way of calculating the correlation level of a system could be by finding,

($\forall$ X, Y Avg. P (Nodes X&Y are down)) / ($\forall$ Z Avg. P (Node Z is down))

Note that this value is not necessarily equal to the average conditional probability. This is due to the fact that, in calculating the two values, averages are taken in different places. Recall that,

Avg. Conditional Probability = Avg. (P (Node X is down | Node Y is down))

= Avg. (P (Nodes X&Y are down) / P (Y is down))

We propose that either one of these can be used as the correlation level. A heuristic is to use the maximum of the two values.

The correlation level is an indication of 2-way correlations. The remaining question is how to determine higher-levels of correlations: the values of 3, 4, … of the storage nodes being correlated. The correlation level, as it is defined here, does not determine these values. The example in Table 4-2 illuminates this fact. In the example, there are two cases regarding the states of nodes A, B and C. In both of the cases, the average conditional probability (i.e., the correlation level as defined above) is 0.5. But, in the first case,

the probability that the three nodes are down simultaneously is greater than zero while in the second case this probability is equal to zero.

[Notation: U- up D- down, assume a period of 6 time units, so in case 1 node A is, up-up-down-down-up-up]	
Case 1	Case 2
A UUDDUU	A UUDDUU
B UUDUDU	B UUDUDU
C UUDUUD	C UUUDDU

Table 4-2. Example – Difference in P (Node A down | Node B and C down)

The new model has an empirical value for P (A node is down | Another node is down) (i.e., the correlation level), but it does not have a value for P (A node is down | Two other nodes are down). To determine the availability of schemes with $n>2$, the model has to estimate the higher order conditional probabilities. The next section describes the model based on conditional probabilities in more detail and explains how this estimation is done.

The Model

The availability model we suggest has 4 parameters: *n*, *m*, *average availability* and the *correlation level*. As an example consider the tree-like structure shown in Figure 4-4. This structure is an easy way of viewing conditional probabilities. We number the nodes from 1 to N, but note that the model assumes that all of the storage nodes are identical.

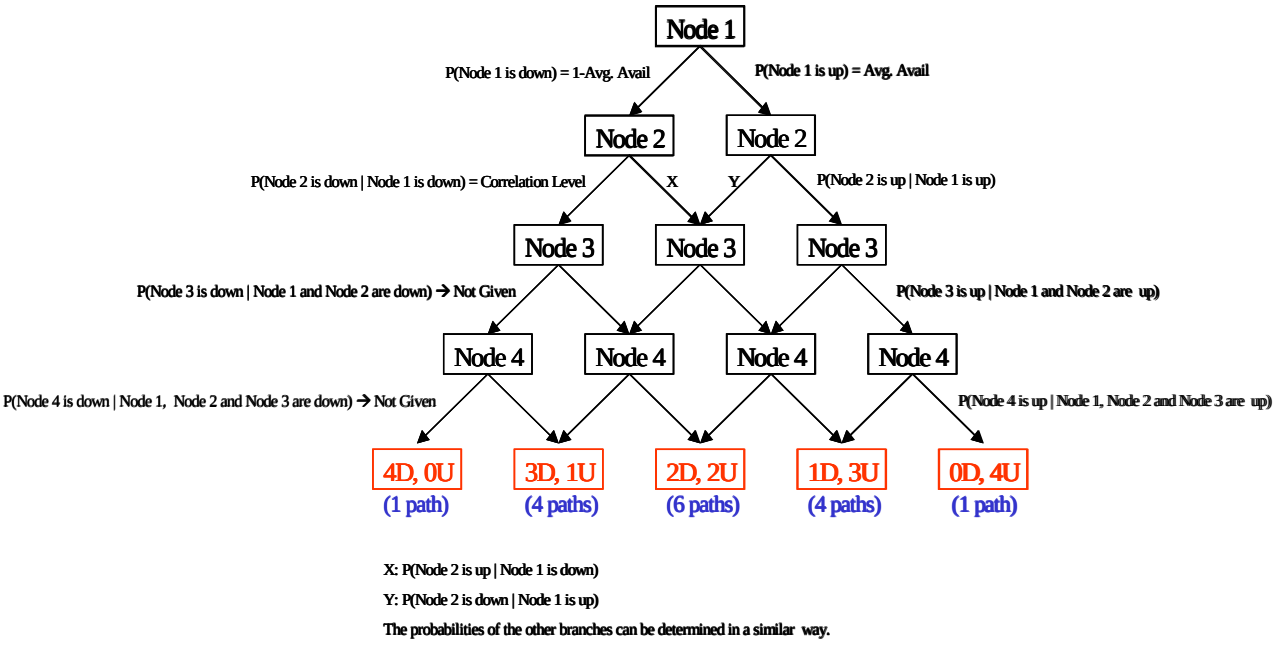


Figure 4-4. The Availability Tree

Structure of the Tree

In the tree, the left branch of each node is the probability that the node is unavailable given the path that has been traversed to reach that node. The probability on the right branch is the probability that the node is available. The leaf nodes represent all the possible cases given 4 nodes: 4 of them are down, 3 of them are down and 1 of them is up etc. Since it is assumed that all the nodes are identical, all the paths leading to a specific leaf node have the same probability. The probability of a path is found by multiplying the probabilities of the branches along that path. To find the probability of a leaf node, it is sufficient to multiply the probability of any path leading to that leaf by the number of paths that lead to that leaf. Therefore, given a scheme, it is trivial to calculate the availability of the scheme by adding the probabilities of the leaf nodes that make that scheme available. For example, to find the availability of the scheme with n=2 and m=2 we do the following. The scheme is available when both of the nodes is up and this probability is equal to

$$\text{P (Node 2 is up | Node 1 is up) * P (Node 1 is up)} \tag{4.7}$$

Calculation of the Values of the Branches

In the model, we assume we are given the average availability and the correlation level. In the tree, the average availability is the value of the right branch of Node 1 and the value of the left branch is 1-(value of right branch). The correlation level refers to the left branch of the first Node 2. 1 minus this value is the value of the right branch (shown as x on the tree). Given the definition of the tree structure, this means that we can calculate P (Node 1 is down and Node 2 is up),

$$\text{P (Node 1 is down \& Node 2 is up) = x * P (Node 1 is down)} \tag{4.8}$$

Also note that we can now calculate the value of left branch of the second Node 2 (shown as y). This is because we assume all the nodes are the same, which implies that,

$$\text{P (Node 2 is down \& Node 1 is up) = P (Node 1 is down \& Node 2 is up)} \tag{4.9}$$

And since we know the values of P (Node 1 is down and Node 2 is up) and P (Node 1 is up) we can calculate y using the following equation,

$$\text{P (Node 2 is down \& Node 1 is up) = y * P (Node 1 is up)} \tag{4.10}$$

Now that we know the value of the left branch of the second Node 2, we can also find the value of its right branch. At this stage we have calculated all the values of the left and right branches of Node 1 and that of the two Node 2s.

But, we have no way of calculating the value of the left branch of the first Node 3. This is where estimation is needed. Note that the left branch of the first Node 3 is P (Node 3 is down | Node 1 and Node 2 are down).

Estimation

To make a good and generic estimation, we examined our datasets to determine if there were any patterns or relations between the parameters of the model and the values the model has to estimate. First, we introduce the notation:

$$R(k) \;=\; \frac{\left\{\ \forall X,\quad X \subseteq U,\quad |X| = k,\qquad Avg.(P(k\_nodes\_are\_down))\ \right\}}{\left\{\ \forall Y,\quad Y \subset U,\quad |Y| = k-1,\qquad Avg.(P([k-1]\_nodes\_are\_down))\ \right\}} \tag{4.11}$$

$$where \quad R(1) = 1 - Avg.Avail.$$

[U is the set of all nodes in the system.]

We calculated the empirical $R(x)$ values, for x=1,2 … 10, for the datasets described in Section 4.4.2. They are shown as solid lines in Figures 4-5 and 4-6. Figure 4-5 shows the datasets from the Web Servers measurements. Figure 4-6 shows the datasets from the Desktops measurements. Note that the scales of the y-axes are different for the two figures. The dotted lines are the model's values. The *model*'s values are the probabilities shown on the leftmost branches of the tree,

$$\begin{aligned} R(1) &= P(Node\,1\_is\_down) \\ R(2) &= P(Node\,2\_is\_down \mid Node\,1\_is\_down) \\ R(3) &= P(Node\,3\_is\_down \mid Node\,1\_\&\_Node\,2\_are\_down) \\ &etc. \end{aligned} \tag{4.12}$$

Note that, the equalities in the equations of $R(2)$ and $R(3)$ hold due to the assumption in the model that all the nodes are identical. The two input parameters, average availability ($R(1)$) and correlation level ($R(2)$), are marked for Dataset-1 shown in Figure 4-5. The model has to estimate all the other $R(x)$ values (x=3, 4 … 10). The difference between $R(2)$ and $R(1)$ is an indication of the level of 2-way failure correlations. When this difference is zero there is no 2-way correlation and if this is the case then estimating that,

$$R(3) = R(2) = R(1) \tag{4.13}$$

would be reasonable. On the other hand, when this difference is positive, this means that failures are correlated.

Both of the datasets shown in Figures 4-5 and 4-6 (solid lines), in general, indicate that $R(3)$ values are higher than $R(2)$ values. For most of the lines it is also the case that $R(x)$s are higher than $R(x\text{-}1)$s. Now that we have identified the increase we have to determine how this increase happens. First of all, we want to make sure that we do not over-fit our datasets, therefore, our formula should be as generic as possible (i.e., we do not want a fourth or fifth order polynomial equation). We propose the following estimation,

$$R(x) = R(x-1) + \frac{[R(x-1)-R(x-2)]}{2} \quad for \quad x = 3,4,...10 \tag{4.15}$$

which says the difference between $R(x)$ and $R(x\text{-}1)$ decreases *exponentially*, as x increases. To ensure the estimated probability does not exceed 1, we add the following term to the previous formula,

$$R(x) = \min\left(R(x-1) + \frac{[R(x-1)-R(x-2)]}{2} \;,\; \frac{R(x-1)+1}{2} \right) \quad for \quad x = 3,4,...10 \tag{4.16}$$

Figures 4-5 and 4-6 show that the estimations obtained (dotted lines), using the above formula, are lower than the actual values (solid lines). The lines that show the estimations tend to flatten out earlier. We describe some of our attempts to overcome this problem. In the discussion, we use $D(x)$ to represent the difference between $R(x)$ and $R(x\text{-}1)$.

The first attempt to correct the problem was to change the estimation in a way that $D(x)$ decreases *linearly* after x>4. The problem with this approach is that for Dataset-1 shown in Figure 4-5 (Web Servers), the estimations become high compared to the real values. Also the estimations for the datasets, which we thought would benefit from this approach, did not improve much. For example, for the Desktops datasets shown in Figure 4-6, the problem is that since initially $D(x)$ decreases exponentially, $D(4)$ becomes very low. Thus, changing to a linear decrease, after x>4, does not provide much benefit. For Dataset-3 and Dataset-4 shown in Figure 4-5, the input parameters (i.e., $R(1)$ and $R(2)$) are very low compared to $R(x)$, where x≥6. Thus, capturing this behavior in the estimations is difficult.

The second attempt to correct the flattening problem was to change only the way $R(3)$ is estimated. The rationale here is to have a very accurate estimate for $R(3)$ from which the further estimations can benefit. A way to do this is by estimating $R(3)$ using:

$$R(3) = R(2) + \frac{[R(2) - R(1)]}{(4/3)} \quad (4.17)$$

This heuristic provides a slightly better estimation compared to the original method, but it lacks the generality of that method. Thus, it is possible that this results in over-fitting.

One final heuristic could be to use $R(4)$ as the correlation level. The problem with this approach is that such a parameter is not easy to reason about and once again it could be that we are over-fitting our data. Note that we have access to only two datasets. Having more datasets would enable us to make a more generic estimation.

Substituting the Estimation into the Tree

Now that we know how to calculate all the $R(x)$'s, we can substitute them into the tree and we can calculate the values of the branches of all the nodes (in a similar way we did for the branches of the Node 2's). Note that the model reduces to the classic model when Correlation Level = (1-Avg.Avail.).

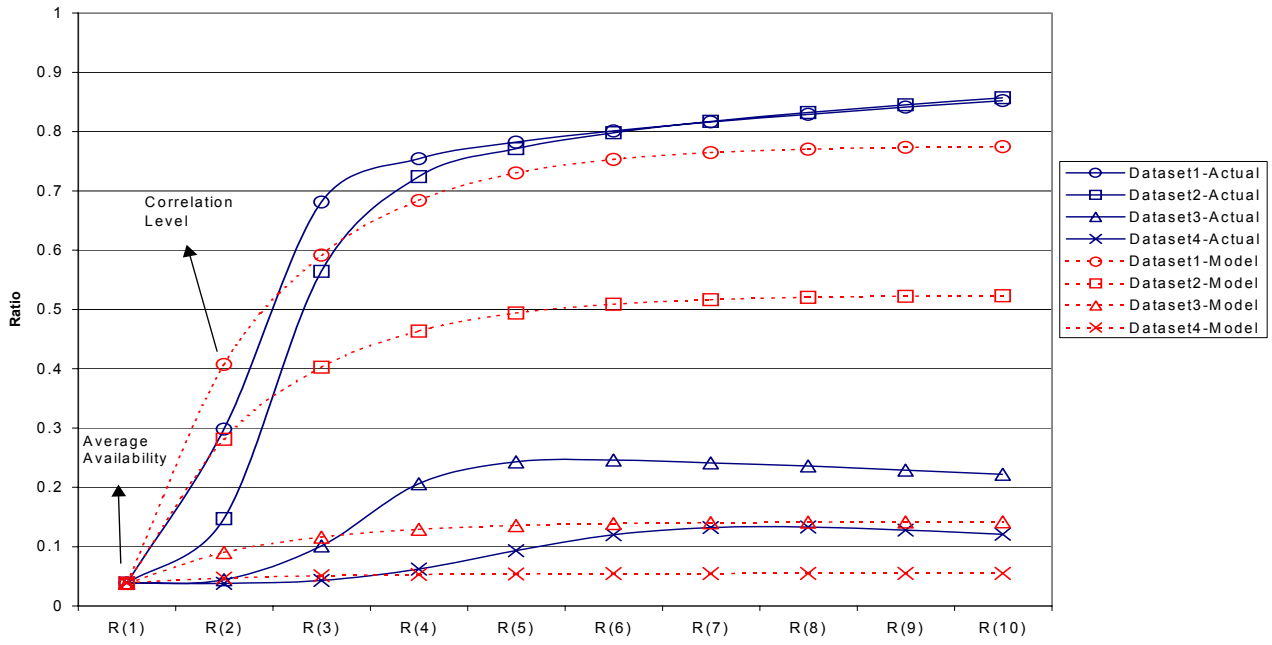


Figure 4-5. Each line represents a different subset of the entire Web Servers dataset. Each dotted line is the estimation of the model for the corresponding solid line (e.g. the dotted line with triangles corresponds to the solid line with triangles). The average availability and correlation level for Dataset1 is shown on the figure. R(x) is defined in the text.

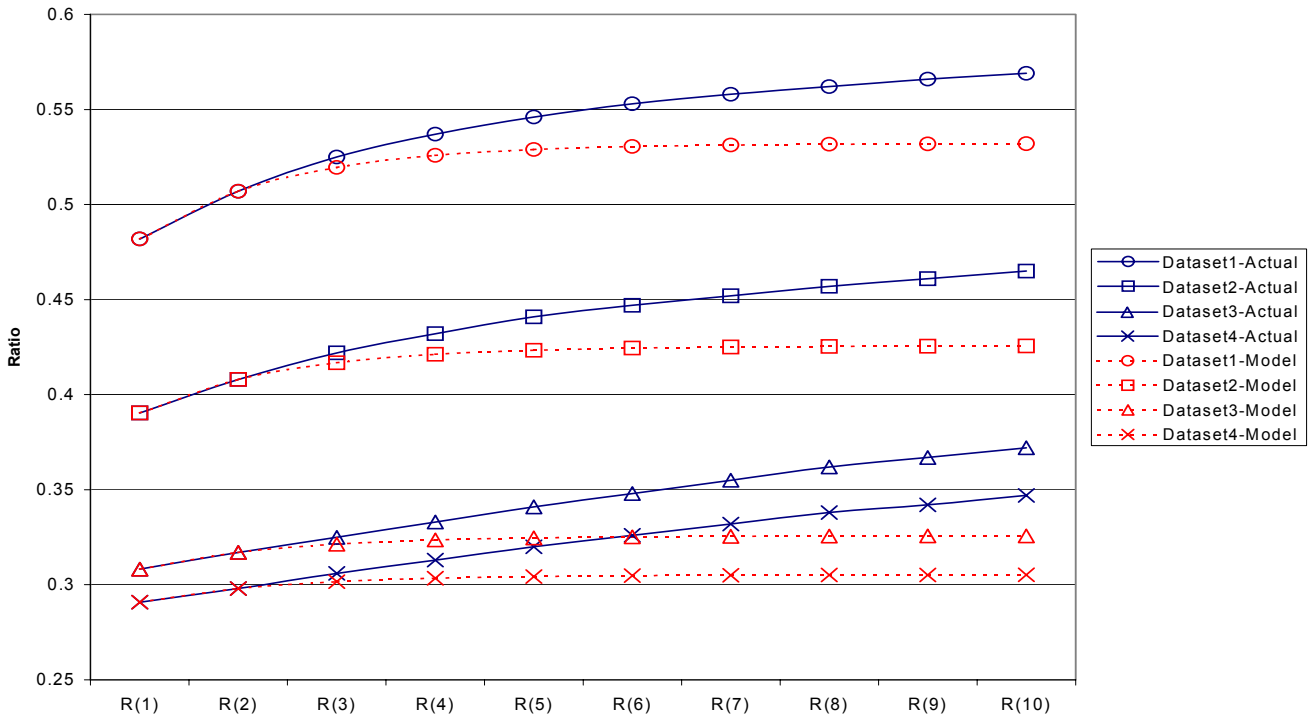


Figure 4-6. Same as the above figure except that the data here comes from the Desktops dataset. Note that the y-axis here is from 0.25 to 0.6.

An Example

Using the tree let us calculate the availability of the scheme with n=4 and m=3. For this scheme to be available there has to be 3 or more nodes that are available. Therefore,

$$\text{P (Avail.)= P (4 up) + P (3 up, 1 down)} \tag{4.18}$$

Recall that every path from Node 1 to a leaf node is a possible scenario. The probability of a path is calculated by multiplying the probabilities on the branches leading to that leaf. The paths that lead to the same leaf node have the same probability. Thus, the probability of the leaf node can be calculated by multiplying the value of any path that leads to that node by the number of paths that lead to that node. For the case of P (3 up, 1 down) there are 4 paths that lead to the leaf node labeled [3U, 1D]. For P (4 up) there is only one path.

4.3.3.4 The Beta-Binomial Distribution

The second model we suggest for modeling the availability of correlated failures is based on the Beta-Binomial distribution. The Beta-Binomial distribution has been used to model other correlated events [Griffiths73, Nicola90]. The distribution is similar to the model based on conditional probabilities in the sense that it uses average availability and a correlation level as parameters. The Beta-Binomial distribution is computed by randomizing the parameter (p=failure probability) of the Binomial distribution. In general the randomized Binomial distribution is,

$$b_N(i) = \int_0^1 \binom{N}{i} p^i (1-p)^{N-i} f_p(p)\,dp \tag{4.19}$$

The intensity function $f_p(p)$ gives the probability distribution that a fraction, *p,* of all nodes fail. It is a unit impulse at Average (*p*) when there is no correlation. For the Beta-Binomial distribution, the intensity function is,

$$f_p(p) = \frac{p^{\alpha-1}(1-p)^{\beta-1}}{B(\alpha,\beta)}, \quad 0 < p < 1, \quad \alpha,\beta > 0 \tag{4.20}$$

where $B(\alpha,\beta)$ is the beta function,

$$B(\alpha,\beta) = \int_0^1 p^{\alpha-1} \quad (1-p)^{\beta-1} \quad dp \tag{4.21}$$

Finally, the formula of the probability that i out of N machines fail is

$$b_N(i) = \binom{N}{i} \frac{(\pi+(i-1)\vartheta)(\pi+(i-2)\vartheta)...\pi(\chi+(N-i-1)\vartheta)(\chi+(N-i-2)\vartheta)...\chi}{(1+(N-1)\vartheta)(1+(N-2)\vartheta)...1} \tag{4.22}$$

$$where \quad i = 0,1,2,...N$$

$$\pi = (1-\chi) = (1-Avg.Avail) = \frac{\alpha}{\alpha+\beta} = \quad Average(p)$$

$$\vartheta = Correlation \quad Level = \quad \frac{1}{\alpha+\beta}$$

In the literature there are different methods for estimating the correlation level from the data [Griffiths73]. In this context, the correlation level is used as a measure of variation in *p*. To compute the correlation level, we adopted the method where first the availability measurements are used to determine the value of $b_2(1) + b_2(2)$ empirically (i.e., the case of n=2, m=2). The measured value of $b_2(1) + b_2(2)$ is used to solve for the unknown correlation level ϑ. Once the correlation level is determined the distribution can be used to estimate the availability of any scheme (*n*, *m*). As an example of using the Beta-Binomial distribution, the availability of the scheme with n=4 and m=2 would be $(1- b_4(3)-b_4(4))$, which means that it can survive at most 2 failures.

The Beta-Binomial distribution reduces to the Binomial distribution (independent failures) when the correlation level is zero. In that sense, the correlation level is different from the correlation level used in the model based on conditional probabilities (i.e., in that model independent failures means Correlation Level = (1-Avg.Avail.)).

4.3.4 Comparison of the Models

In this section, we evaluate our availability models against the classic model using the web server dataset.

Comparison Using the Web Servers Datasets

Figures 4-7, 4-8, 4-9 and 4-10 show the comparison for the Web Servers Dataset-1, Dataset-2, Dataset-3 and Dataset-4, respectively. The graphs have 4 types of bars and show the comparison of the actual availability of each scheme, and the estimations of the model based on conditional probabilities, the Beta-Binomial distribution and the classic model. The figures show that the model based on conditional probabilities and the Beta-Binomial distribution are, in general, more accurate than the classic model. The Beta-Binomial distribution and the model based on conditional probabilities are comparable. On average, the model based on conditional probabilities is more precise.

The average and maximum absolute errors (units are nines of availability) for the Web Servers datasets are listed in Table 4-3. For all four datasets, the model based on conditional probabilities outperformed the other two models. A point to note is that, although the maximum error of the model based on conditional probabilities for dataset 4 is ~2 nines, this error occurs for the scheme with n=10 m=1. The actual availability of this scheme is ~11 nines (0.00031536 seconds unavailability per year) the estimated availability is ~13 nines (0.0000031536 seconds unavailability per year). Therefore, looking for in which scheme the error has occurred is more informative.

	Model Based on Cond. Probabilities		Beta-Binomial		Classic Model	
	Avg. Error	Maximum Error	Avg. Error	Maximum Error	Avg. Error	Maximum Error
Dataset-1	0.127	0.297	0.229	0.717	2.951	11.407
Dataset-2	0.32	1.28	0.529	2.25	2.671	11.001
Dataset-3	0.32	1.202	0.942	3.781	1.406	5.940
Dataset-4	0.32	1.961	0.711	3.777	0.56	3.124

Table 4-3. Absolute Errors for the Web Servers (Unit: nines of availability).

We now analyze the error of the model based on conditional probabilities in detail. Figure 4-11 shows only the schemes with m=1 for Dataset-1. The error for the scheme n=2 m=1 is due to the fact that the model uses only averages and does not consider the variance of the availabilities and the variance of the correlations of the servers. The following example illustrates the problem. Assume there are only 2 servers in the system, the average availability is 0.95 and the correlation level is 0.5. The model's estimation of the availability for the scheme n=2 m=1 is:

= 1 - P (both servers are down)

= 1 - 0.05 * 0.5

= 0.975

Further assume that, in reality, one of the servers has an availability of 0.975 and the other has an availability of 0.925 (the average is 0.95). In this case, assuming the correlation level is 0.5, the two conditional probabilities can be calculated as follows (i.e. Corr_1 = P (A is down | B is down) and Corr_2 = P (B is down | A is down)):

0.025 * Corr_1 = 0.075 * Corr_2 (the portion of the downtimes that overlap should be equal)

(Corr_1 + Corr_2)/2 = 0.5 (0.5 = Correlation level)

Therefore, Corr_1 = 0.75, Corr_2 = 0.25 and the actual availability is;

= 1 – 0.025 * Corr_1 (which is equal to 1- 0.075 *Corr_2)

= 0.98125

which is higher than the model's prediction.

A more detailed discussion of this issue can be found in the next section. For the schemes with m=1, if n is low, the model under-estimates, but it over-estimates for higher n's. This increase is because the model under-estimates the higher x-way failure probabilities.

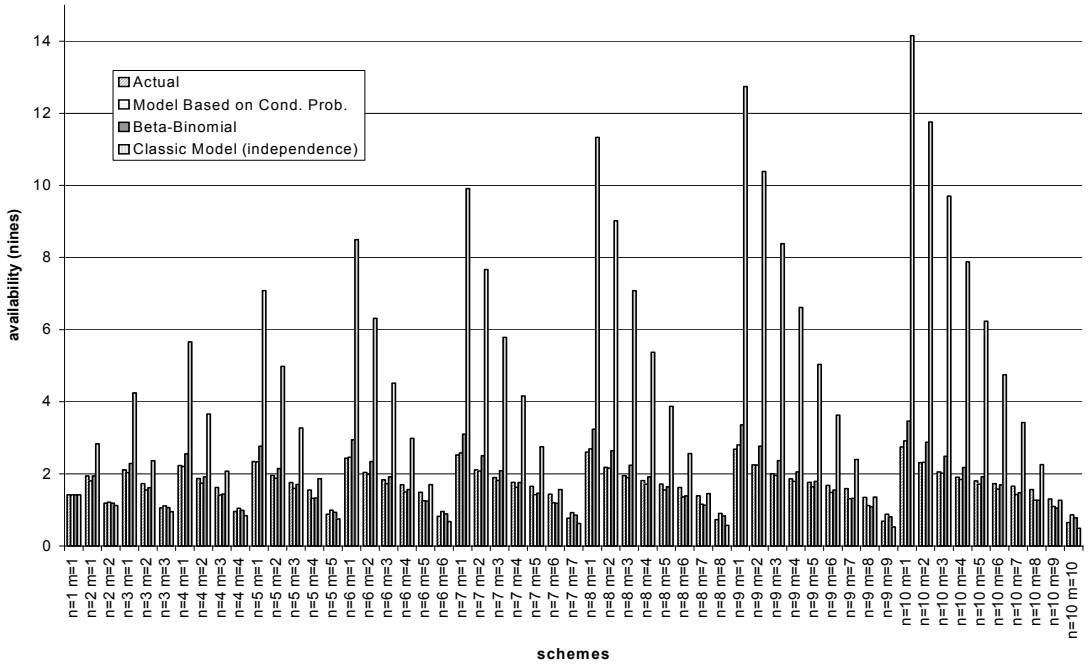


Figure 4-7. Comparison of the 3 Models using Web Servers Dataset-1

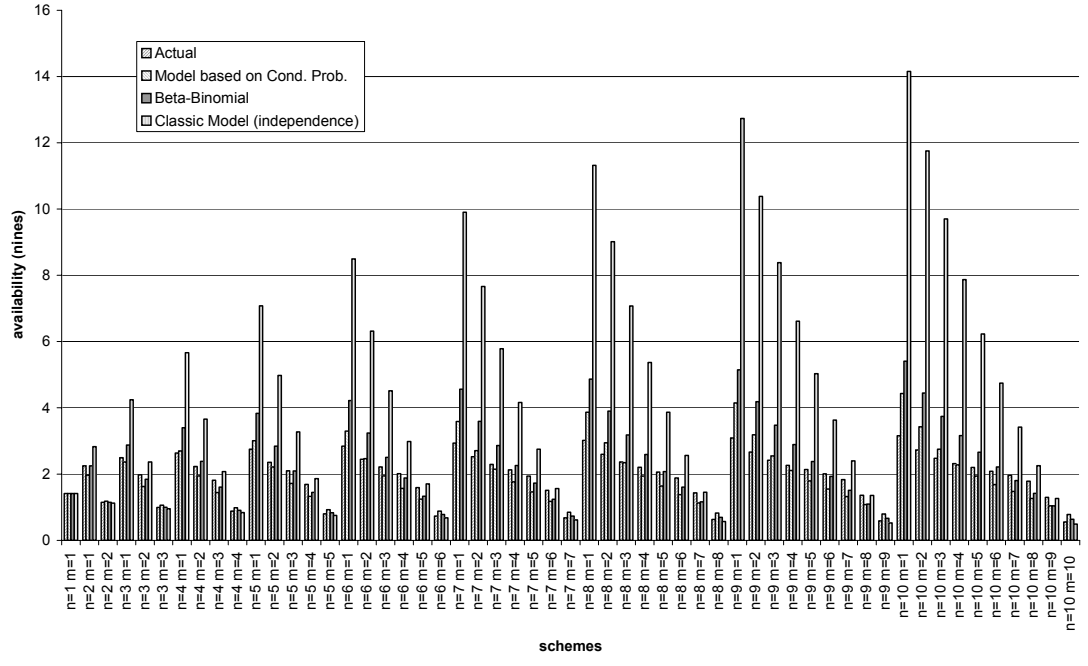


Figure 4-8. Comparison of the 3 Models using Web Servers Dataset-2

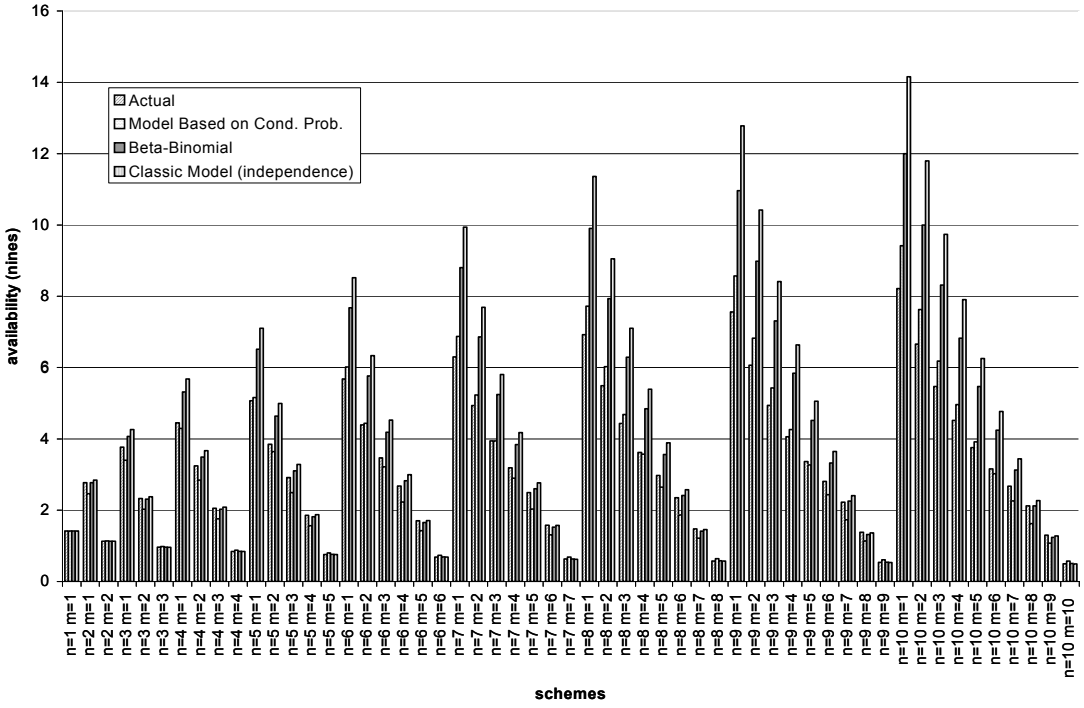


Figure 4-9 Comparison of the 3 Models using Web Servers Dataset-3

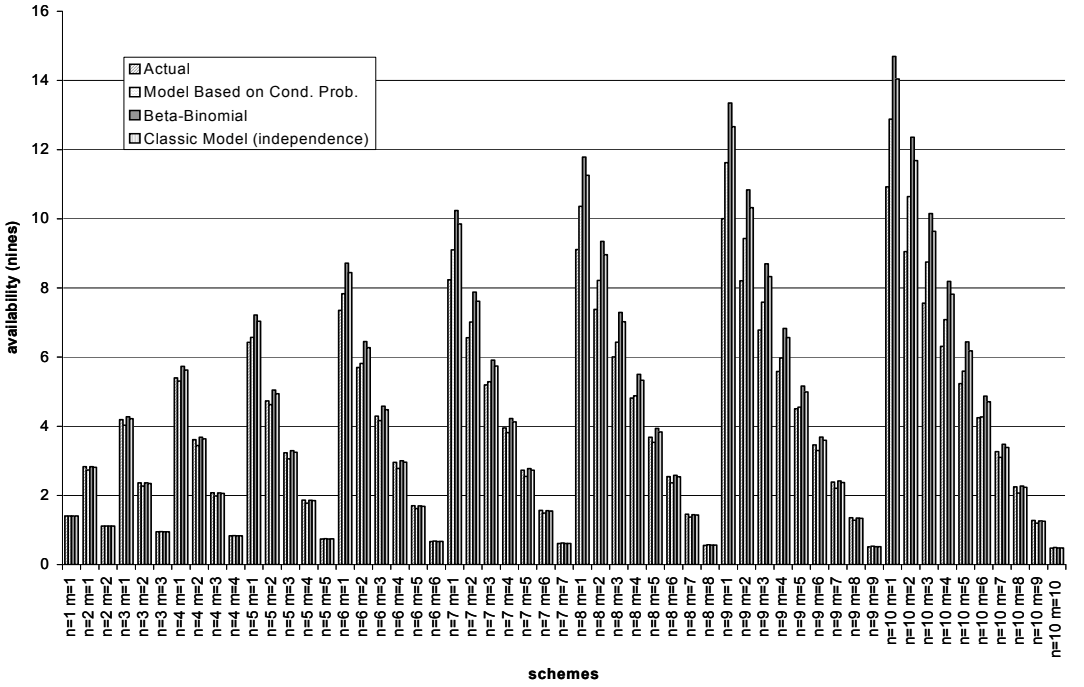


Figure 4-10 Comparison of the 3 Models using Web Servers Dataset-4

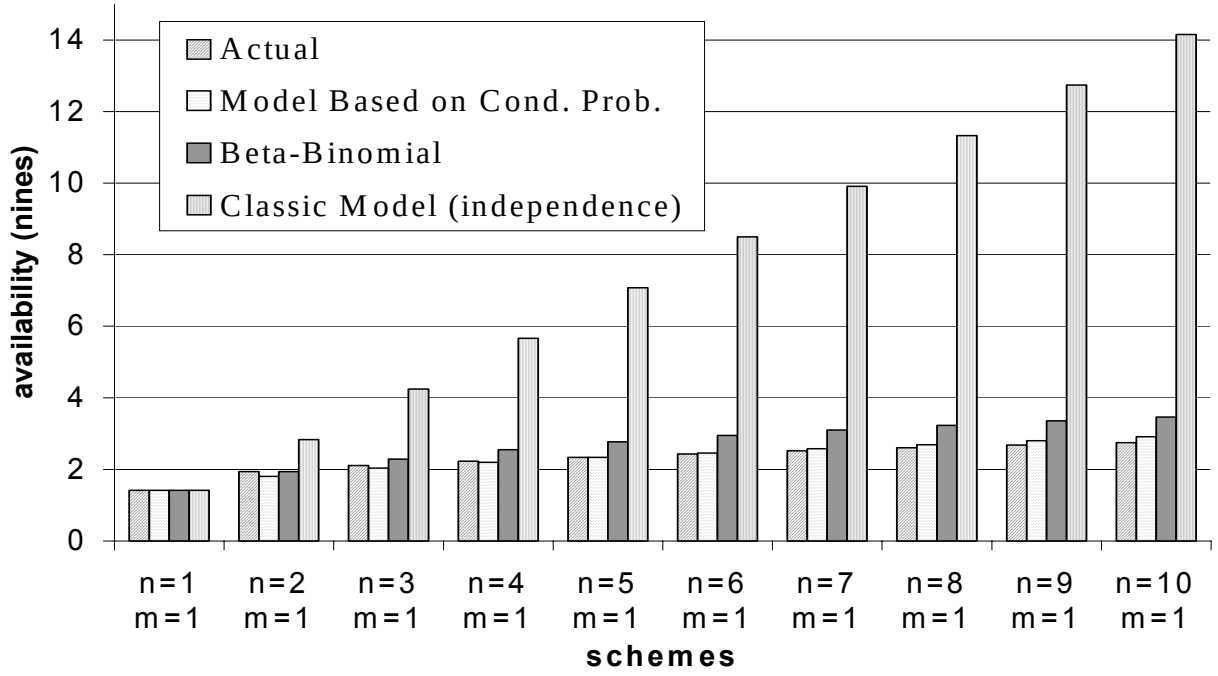


Figure 4-11. Availabilities of Schemes with m=1 for the Web Servers Dataset-1

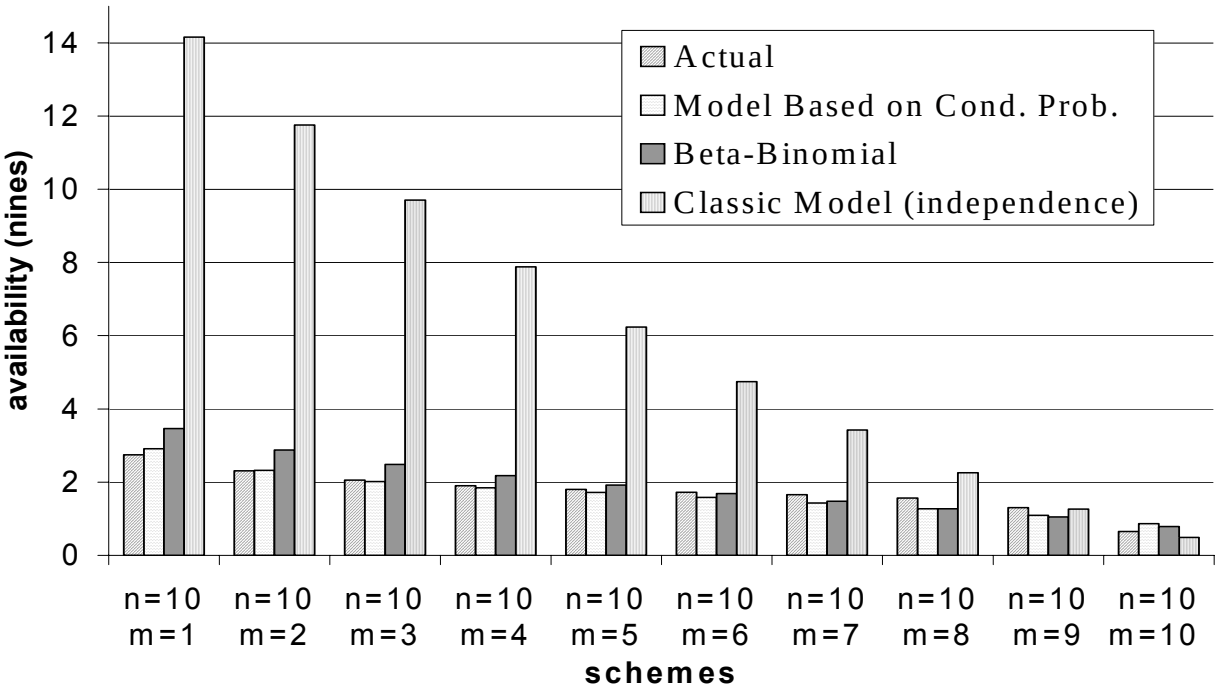


Figure 4-12. Availabilities of Schemes with n=10 for the Web Servers Dataset-1

In Figure 4-12, the schemes with n=10 for Dataset-1 are shown. For m between 1 and n-1, the model first over-estimates and then underestimates. Note that the Beta-Binomial distribution exhibits the same behavior. This is mainly due to two factors. First, note that a decrease in failure correlation increases the availability of schemes with large (n–m)'s more than schemes with small (n–m)'s. Since the model underestimates the correlation, availability of schemes with large (n–m)'s tend to be over-estimated. Second, the model assumes that all the nodes are identical and assumes smooth transitions from one x-way correlation to another. In reality, some sets of servers are more correlated to each other than to others; thus, using average correlation and average availability leads to inaccuracies.

For schemes with n=m, the two proposed models always over-estimate. This is because the models consider only the average availability. In reality the schemes with n=m are bound by the server with minimum availability. In summary, a model that uses only average availability is bound to over-estimate the availability of schemes with n=m unless it significantly under-estimates the higher x-way correlations (e.g., independence).

Comparison Using the Desktops Datasets

Figures 4-13 and 4-14 show the comparison of the 4 models using the Desktops Dataset-1 and Dataset-3. The failures of the machines in this set were nearly independent. The importance of using the Desktops measurements in our comparison is to show that the proposed models work for datasets that are nearly independent as well as highly correlated datasets. Both of the models were accurate for this case. The interesting point here is that the Beta-Binomial distribution has different behaviors for the two datasets. The way Beta-Binomial determines correlation is by looking at the actual availability of the scheme n=2 m=2. Therefore, its accuracy is limited by the accuracy of the correlation calculated. Table 4-4 shows the errors of the 3 models for the four datasets.

	Model Based on Cond. Probabilities		Beta-Binomial		Classic Model	
	Average Error	Maximum Error	Average Error	Maximum Error	Average Error	Maximum Error
Dataset-1	0.018	0.148	0.023	0.238	0.086	0.494
Dataset-2	0.023	0.188	0.021	0.205	0.087	0.489
Dataset-3	0.032	0.263	0.001	0.017	0.076	0.456
Dataset-4	0.033	0.259	0.001	0.022	0.072	0.427

Table 4-4. Absolute Errors for the Desktops (Unit: nines of availability).

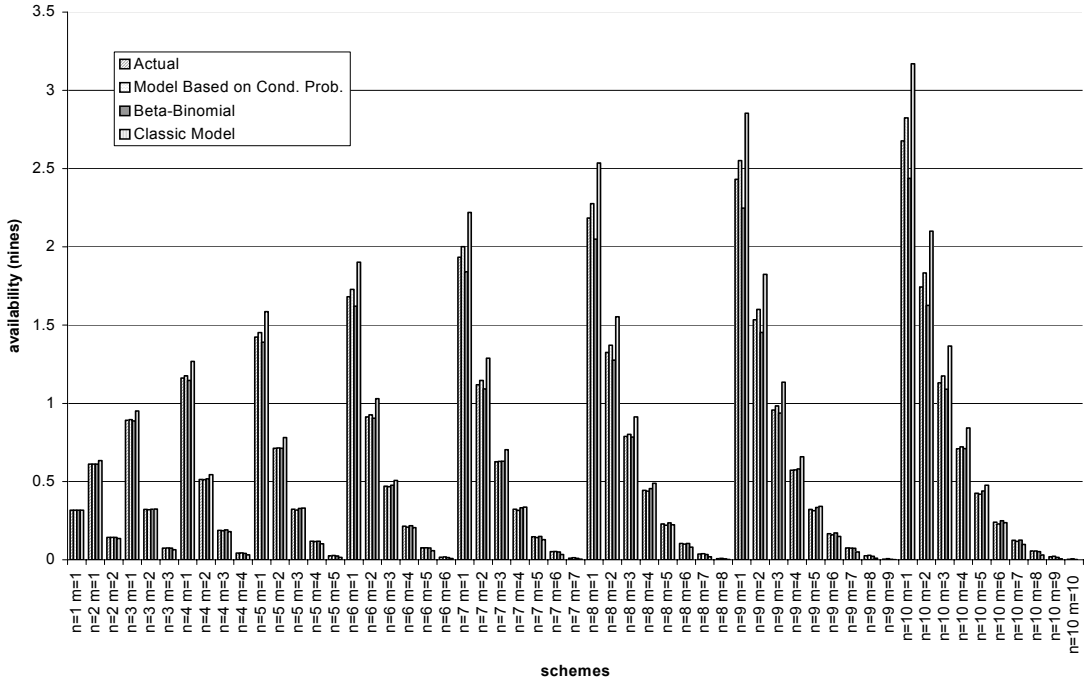


Figure 4-13. Comparison of the 3 Models using Desktops Dataset-1

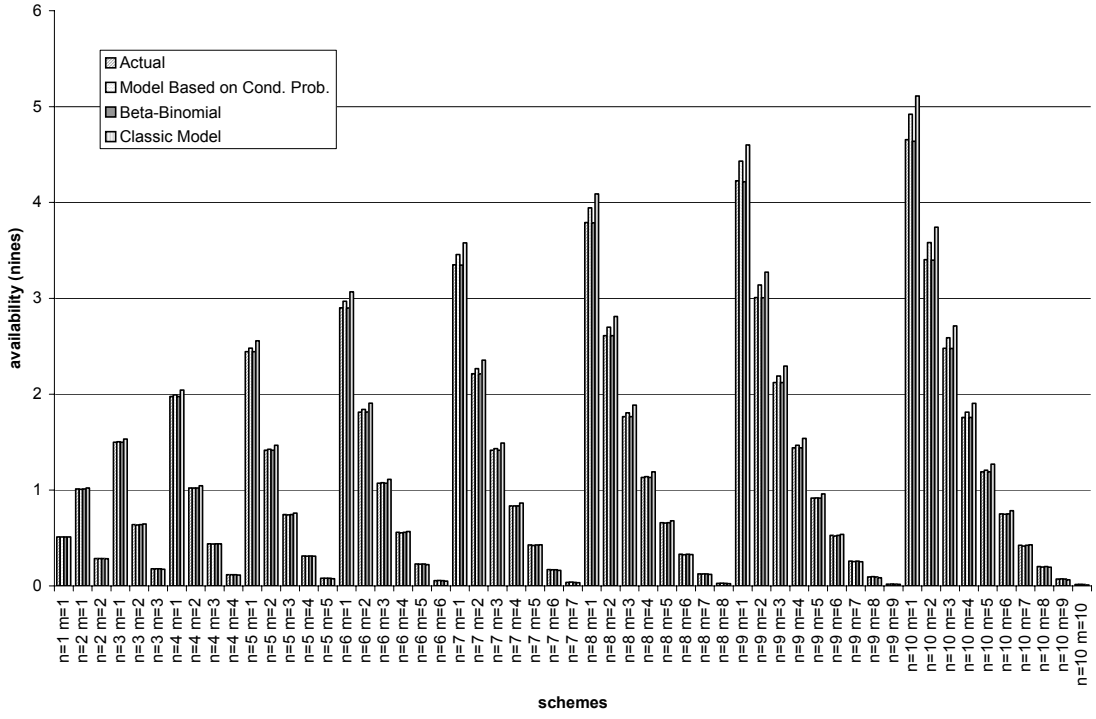


Figure 4-14. Comparison of the 3 Models using Desktops Dataset-3

4.3.4.1 Sensitivity of the Model Based on Conditional Probabilities

In section 4.2, we gave an example that showed that the classic model was unable to cover the range of possible availabilities for the case of two machines with availability 0.99. This was due to the independence assumption. Since the model based on conditional probabilities considers correlation, it is able to account for the full range (see Table 4-5).

However, there are cases when the model is not perfectly accurate. We now analyze the new model in depth.

Correlation Factor	New Model's estimation
perfectly correlated → 1	0.99
independent → 0.01	0.9999

Table 4-5. Effect of having a Correlation Factor

Using Averages

The model uses only one parameter that defines the availability of the nodes: the average availability. It does not consider the variance among the servers' availabilities. A heuristic such as, using the geometric mean instead of the arithmetic mean could be used. The geometric mean is always less than or equal to the arithmetic mean. However, the geometric mean is not any more general then the arithmetic mean. Depending on whether in the dataset the machines with lower availabilities have higher correlation or the machines with higher availabilities have higher correlation either the geometric mean or the arithmetic mean could result in more accurate estimations. Also the accuracies of the two methods depend on what the scheme parameters are. Therefore, no matter how the availability parameter is calculated, the model is agnostic to the variance among the servers and this has an effect on the accuracy of the model's estimations. But note that we do not want to have an extra variance parameter in the model since this would complicate the model.

To support the hypothesis that not considering variances leads to inaccuracies, we conducted a second experiment using the Web Server measurements. In this experiment we used a subset of the servers (14 of them) that have similar availabilities (i.e., low variance). Figure 4-15 shows the graph obtained. In this case the model's estimations are much closer to the actual availabilities. The average error is 0.053 nines and the maximum error is 0.117 nines.

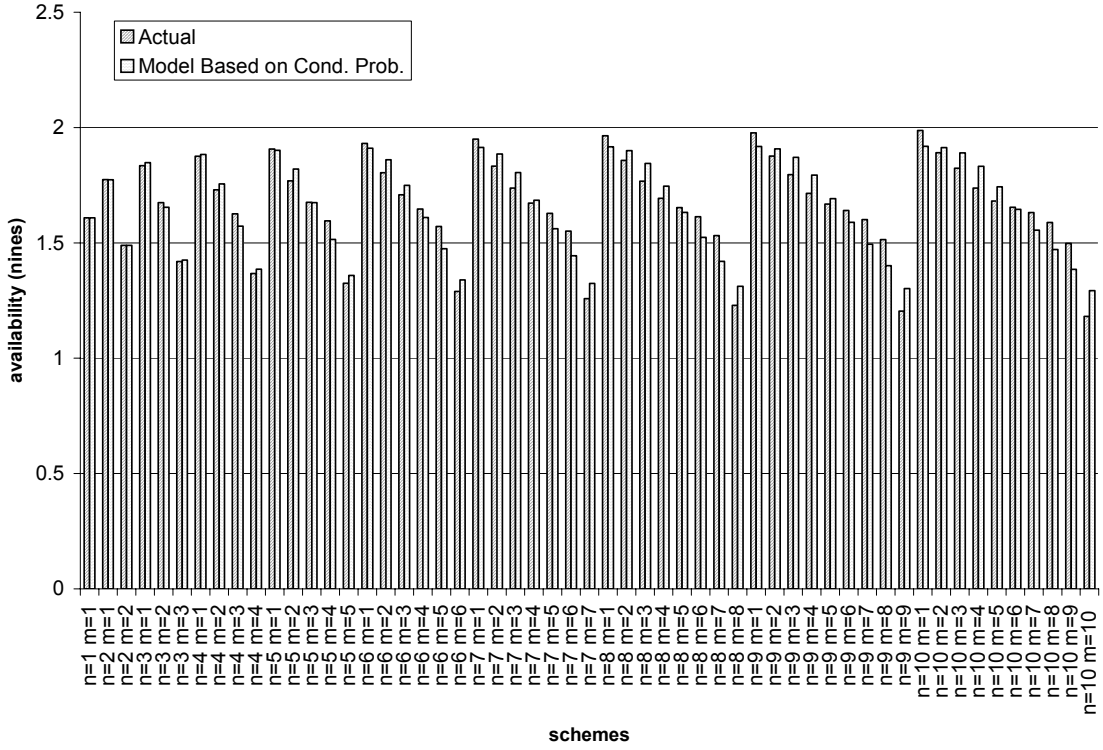


Figure 4-15. Comparison Using a Subset of the Web Servers

Incomplete Correlation Information

As mentioned earlier, the model estimates the higher-level correlations using the two parameters and this estimation can lead to inaccuracies. Our results in Section 4.4 show that the model based on conditional probabilities is accurate for the Web Servers and Desktops datasets. For datasets 3 and 4 from the Web Servers measurements, although the 2-way correlation (i.e. the correlation level) is not representative of the higher x-way correlations, the model's estimations are reasonably accurate. It is conceivable that some cases may have zero 2-way correlation but high x-way correlations. In such a case since the correlation level indicates no correlation the accuracy of the model will be low.

4.3.5 Related Availability Modeling Work

In the literature, there are several approaches to modeling the availability of a single machine. Most of these studies are based on Markov Chain models [Garg99, Kalyanakrishnam99]. For example, in [Kalyanakrishnam99], each state represents a level of functionality of the machine, such as "reboot", "connectivity problems", "adapter problems", "disk problems", "shutdown" etc. and the weight of transitions between states is determined empirically.

There are similar state-based modeling studies for multiple machines. Those techniques are useful for cases where there are a small number of machines and it is possible to list all the states of the entire system. In [Kalyanakrishnam99], they model mail servers using a finite state model. Specifically they have a primary machine and a number of backup machines. They identify all the possible states and the transition probabilities (empirical values) between the states. They account for correlated failures by having a state that represents the case when more than one machine is unavailable. Their model is completely em-

pirical. Such an approach is impractical for cases where there are hundreds of servers since it would be impossible to identify all of the states and the relationships between states.

In [Tang92], they present an approach to correlation modeling that is similar to the model based on conditional probabilities. They also use conditional probabilities as a measure of correlation among failures of machines. They only model 2-way correlations, whereas in our case, to determine the availabilities of schemes with $n>2$, in addition to 2-way correlations higher-way correlations are also modeled. Therefore, their model does not involve or require any estimation of unknown probabilities.

In [Douceur01c], effective system availability (ESA) is used to compare the availabilities of different schemes. The availabilities of individual files are calculated using the independence assumption. ESA is a weighted average of the availabilities of the files in the system. Files with low availabilities have a higher weight. For N files, each with availability a_i (number of nines), ESA is calculated as follows,

$$ESA = -\log \frac{1}{N} \sum_{i=0}^{N-1} 10^{-a_i} \quad (4.23)$$

They also calculate the actual system availability and they compensate for the difference between the actual ESA and the calculated ESA by finding the following relationship [Douceur01c].

$$ESA_{actual} = -0.078\ (ESA)^2 + 1.5\ ESA - 0.84 \quad (4.24)$$

In [Douceur01c], they do not claim that their model is generic, and they mention that this model is valid for ESA values in the range of 2 to 6 nines. When we try their model for our Web Servers dataset, we get errors such as negative numbers and erroneous mappings, such as ESA~14 becomes ~4.8 but ESA~9 becomes ~6.3. Because of such errors and the inaccuracy of their model for our Web Servers dataset, we do not include their model in the comparison.

In [Littlewood89], they model failures in multi-version software. Their approach results in a complicated model that provides little guidance to system designers.

4.4 Ordering of the Schemes

In Section 3, we described how scheme selection is done. In scheme selection, an important point is the ordering of the schemes. The reason why the ordering of the schemes is important is due to the fact that the user could have estimated the average availability of her system incorrectly. Thus, if this effects the ordering of the schemes, the scheme the user selects (e.g., the 50th percentile scheme) using the incorrect estimate would not be the optimal scheme.

4.4.1 Analysis

In this section, we investigate how average availability and the correlation level affect the ordering of the schemes. By "ordering", we mean the schemes sorted according to their availabilities. In our analysis, we use the model based on conditional probabilities and we only consider schemes with $n \leq 10$.

Varying Average Storage Node Availability

Here we consider four cases: varying average storage node availability from 0.90 to 0.95, from 0.90 to 0.99, from 0.90 to 0.999 and from 0.90 to 0.9999. The failures are assumed to be independent. We classify schemes according to their change in rank (i.e., if a scheme is 3rd and becomes 5th, the change in its rank is 2). Table 4-6 shows the number of schemes in each class. The first row shows the number of schemes that have a rank change of 0 (i.e., same rank). The table shows that overall the rank changes are not significant; only a few schemes change significantly. Also note that the ordering stabilizes after 0.999 (3 nines).

	Number of Schemes (In total 55 schemes)			
Change in Rank	From 0.90 to 0.95	From 0.90 to 0.99	From 0.90 to 0.999	From 0.90 to 0.9999
0	**29**	**16**	**16**	**16**
1	**18**	**18**	**14**	**14**
2	**8**	**12**	**13**	**13**
3	**0**	**6**	**6**	**6**
4	**0**	**1**	**2**	**2**
5	**0**	**2**	**2**	**2**
6	**0**	**0**	**2**	**2**

Table 4-6. Changes that result from varying the availability level

Now, we analyze the significance of changes in ranks. Assume the ordering of the schemes using the correct value for average node availability is A, and the ordering of the schemes using an incorrect value for average node availability is B. In ordering B, the scheme at rank x (scheme B) may be different from the scheme at rank x (scheme A), in ordering A. The absolute difference (D) in the availabilities of the two schemes, in the correct setup, defines the significance of the change in ranks. The difference between the two orderings A and B is the average of all D's over all x (x=1, 2 … 55). For the four cases, defined above, we assume 0.90 is the incorrect average node availability. The statistics, for the four cases, are listed in Table 4-7. As the difference between the correct and the incorrect average node availability increases, the significance of the difference increases.

Ordering B (Incorrect Average Storage Node Availability)	Ordering A (Correct Average Storage Node Availability)	Average (D) (nines)	Maximum (D) (nines)
0.90	0.95	0.08	0.36
0.90	0.99	0.39	1.31
0.90	0.999	0.81	2.30
0.90	0.9999	1.24	3.30

Table 4-7. Differences Between Orderings

To discover trends in the ordering of schemes with respect to average storage node availability we performed another experiment. Figure 4-16 shows how the availability for each scheme changes as average availability increases. For clarity, only a carefully chosen subset of the schemes is shown. Note that there are patterns in the change of ranks. For low availability levels, schemes with (n_1, m_1) where m_1=1 are better than schemes with (n_2, m_2) where $n2=n1+x+y$, $m2=m1+x$ (x>0 and y>0). But as availability increases, schemes with (n_2, m_2) start outperforming schemes with (n_1, m_1). For example, the scheme with n=4 m=1 eventually is surpassed by schemes n=10 m=4, n=8 m=3, n=9 m=4, n=10 m=5, n=6 m=2, n=7 m=3, n=8 m=4, n=9 m=5, n=10 m=6. The common property among these schemes is that the difference (n-m) for all of these schemes is greater than that for the scheme with n=4 m=1, where (n-m)=3. In general, schemes with a larger (n-m) eventually outperform schemes with a lower (n-m). For example, the scheme with n=10 m=4 eventually outperforms the schemes with n=5 m=1 and n=6 m=1. This observation also shows that treating (n/m) as the replication factor in terms of the availability is incorrect.

Also, among the schemes that have the same (*n*-*m*), the ones that have lower *n*'s are the winners. That is, *n*=4 *m*=1 outperforms *n*=5 *m*=2. Another observation is that, after 3 nines of average storage node availability, the schemes that have the same (*n*-*m*) form a cluster. In other words, the schemes converge to a total ordering. Within a cluster the ones with lower *n*'s have higher availabilities. The final observation here is that, although there are rank changes between two average storage node availabilities, the absolute scheme availability changes are not large. However, large changes in average storage node availability may result in significant differences.

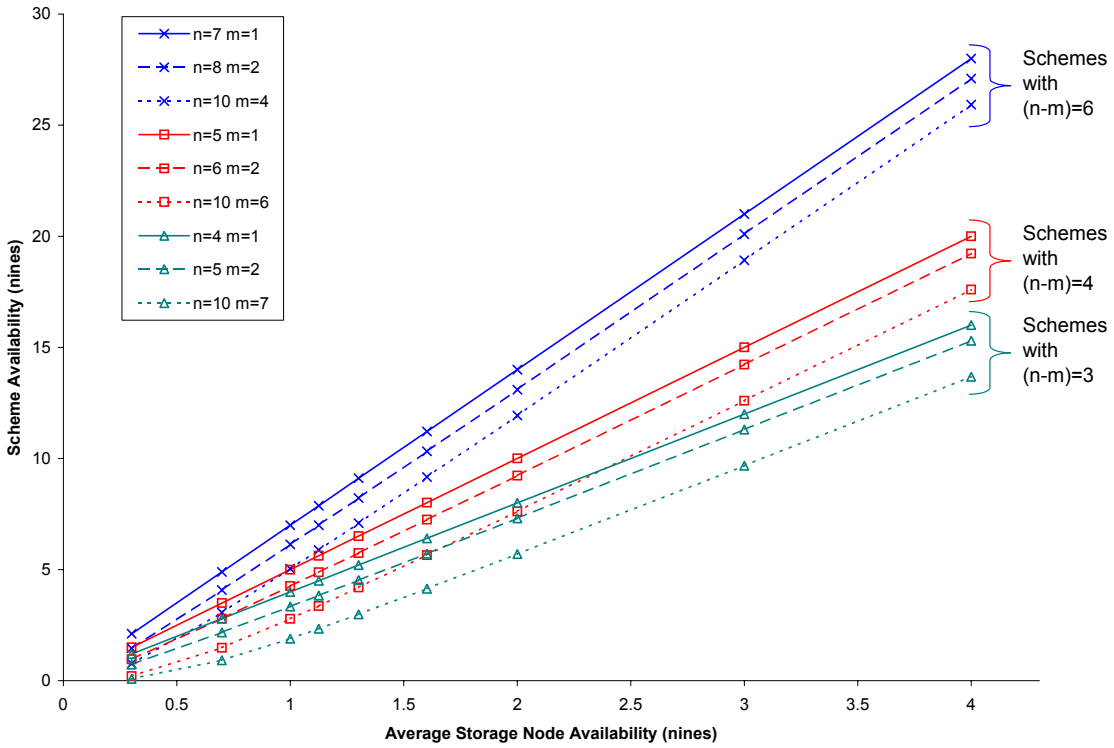


Figure 4-16. Scheme Availabilities as Average Storage Node Availability changes

Varying the Correlation Level

Next, we examined the effect of the correlation level on the ordering of the schemes. We considered two cases: the first case is when the correlation level changes from 0.05 to 0.1 and the second case is when it changes from 0.05 and 0.25. Storage node availability is 0.95 (1.3 nines). We classify schemes according to their change in rank (i.e., if a scheme is 3rd and becomes 5th, the change in its rank is 2). Table 4-8 shows the number of schemes in each class. The first row shows the number of schemes that have rank change 0 (i.e., same rank). Compared to the previous case, there are more changes in ranks. This is mainly because when there is correlation most schemes have a lower availability, and the availabilities of different schemes are closer to one another.

To analyze the significance of the changes in ranks we performed an experiment similar to the one we did for the previous case. The statistics are shown in Table 4-9. Note that, the differences here are less than the previous case. But, since the actual availabilities (Figure 4-17) in this case are less than the previous case, the differences here are more important than the previous case.

	Number of Schemes (In total 55 schemes)	
Change in Rank	From 0.05 to 0.1	From 0.05 to 0.25
0	**18**	**11**
1	**24**	**12**
2	**10**	**7**
3	**2**	**10**
4	**1**	**7**
5	**0**	**3**
6	**0**	**2**
7	**0**	**1**
8	**0**	**2**

Table 4-8. Changes that result from varying the correlation level

Ordering B (Incorrect Correlation Level)	Ordering A (Correct Correlation Level)	Average (D) (nines)	Maximum (D) (nines)
0.05	0.1	0.10	0.42
0.05	0.25	0.17	0.56

Table 4-9. Differences Between Orderings

To discover trends in the rank changes, we examined individual scheme availabilities as the correlation level changes from 0.05 (being independent) to 0.50. Figure 4-17 shows the corresponding graph. For clarity purposes, the graph shows only the availabilities of schemes with (n-m) equal to 1, 3 and 6. The observations drawn are

- The schemes that have the same (n-m) are always ordered according to their m values (the scheme with m=1 being the best).
- The schemes with larger (n-m) are influenced significantly by correlation. Also not shown in the graph, the availabilities of schemes with n=m increase as the correlation level increases.
- As correlation approaches 1, the availabilities of all of the schemes converge to the average storage node availability (1.3 nines in this case).
- The schemes with low (n-m)'s and low m's outperform schemes with large (n-m)'s and large m's. For example, the scheme with n=2 m=1 passes the scheme with n=10 m=7 (Also, not shown in the graph, it passes the schemes with n=7 m=4, n=10 m=6, n=10 m=5 etc.). This is because schemes with large (n-m)'s, are affected more from correlation.

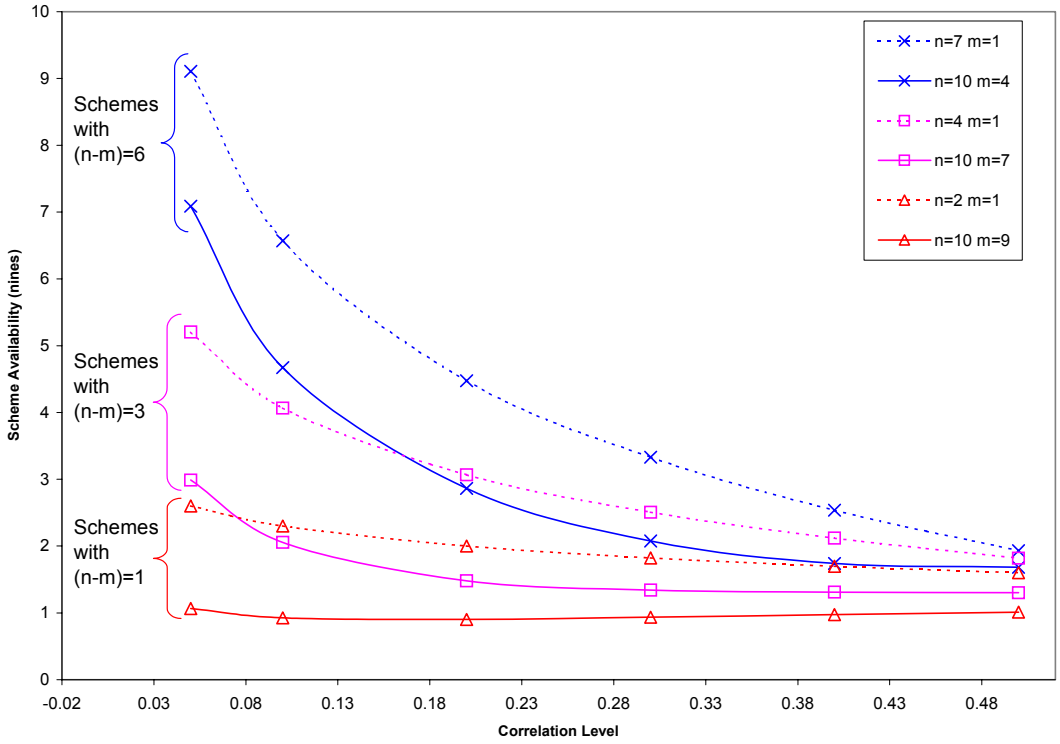


Figure 4-17. Scheme Availabilities as Correlation Level changes

4.5 Placement of Related Files

In designing a distributed system, the ways in which files (shares) are stored may affect the availability of the files. Previous work in databases analyzes how to store the stripes of the primary and the backup copies. Striping is a scheme with parameters $m=n$ and $p=1$ (e.g., $n=3$ $m=3$ $p=1$). The two well-known placement methods are chained de-clustering and interleaved de-clustering [DeWitt90]. The idea in these two methods is to store the corresponding stripes of the primary and backup copies on different machines in a manner that in the face of failures all the data will still be available and the load will be balanced among the remaining nodes. In their case, all of the machines are identical and they only have to store the primary and backup copies of the database. In [Douceur01b], they use a set of desktop machines with different availabilities and their measurements indicate that the failures between machine failures are nearly independent. Since an exhaustive search of the best placement of files is not feasible, they compare three different algorithms, which achieve near optimal placement solutions.

An interesting problem with file placement that is overlooked in the literature is the placement of files that are related. Two files are related if one file can be accessed only if the other file is available. In systems that have a directory structure, a file can only be accessed by traversing the corresponding directory path. In PASIS [Wylie00], directories are stored the same way files are stored (i.e. using threshold schemes). In order to get to a file, all of the directories on the path have to be available. To achieve optimal overall availability, the shares of the related files should be stored on the same set of storage nodes.

4.5.1 Analysis

Independent Failures

In this section, the placement of related files is analyzed in depth. First, we examine the case when failures are independent. Assume, there are N related files, the availabilities of the storage nodes in the system are equal to (Avail.) and the scheme used is n=1 m=1. One extreme case is to store all the files on the same storage node. In that case, the overall availability is equal to the availability of a single storage node. The other extreme case is to store each file on a separate node (as long as there are sufficient number of nodes in the system). The overall availability in that case is $(\text{Avail.})^N$. The difference in availability between the two cases is,

$$(\text{nines (Avail.)} - \text{nines } ((\text{Avail.})^N)) \tag{4.25}$$

where nines is the function that converts the corresponding availability value to nines. For N=100 and Avail.=0.95, the difference is equal to 1.298 nines. Now, let us look at different schemes.

In Figure 4-18, we assume that average storage node availability is 0.95 and the storage node failures are independent (we later relax this assumption). The x-axis shows the number of related files stored and the y-axis shows the difference in availability between the two extreme placement strategies. The first strategy is to store the shares of k files on the same set of n storage nodes. The second strategy is to store the shares of k files on completely different sets of nodes (i.e. on a total of n*k storage nodes). The former always offers better availability and the graph shows the difference between the two strategies as the number of related files (k) increases. For the scheme with n=1 m=1, the difference approaches 1.3 nines, as the number of related files increases. For the scheme with n=10 m=1, the difference approaches 13 nines. This large difference is mainly due to the independence assumption. For the schemes with n=m, since the availability achieved is lower the line tends to flatten earlier. For the scheme with n=10 m=10, the maximum difference is ~0.4.

Correlated Failures

Now, let us examine the same problem when the failures between storage nodes are correlated. First, using our model, we show how the availability is calculated when the shares of different files are stored on distinct sets of storage nodes.

Consider the scheme *n*=2 and *m*=1 and assume there are 2 files stored on a total of 4 storage nodes (Figure 4-19). The shares of one of the files are stored on Node 1 and 2. The shares of the other file are stored on Node 3 and 4. If 3 or more nodes that are available, then the 2 files are available. If there are only 2 nodes available nodes, then for the two paths, shown in the figure, the files are unavailable. Note that, this is different from just having 2 of the 4 nodes up. Using this method, we analyzed the related files issue for the case when the average storage node availability is 0.95 and the correlation level is 0.25. The corresponding graph is shown in Figure 4-20.

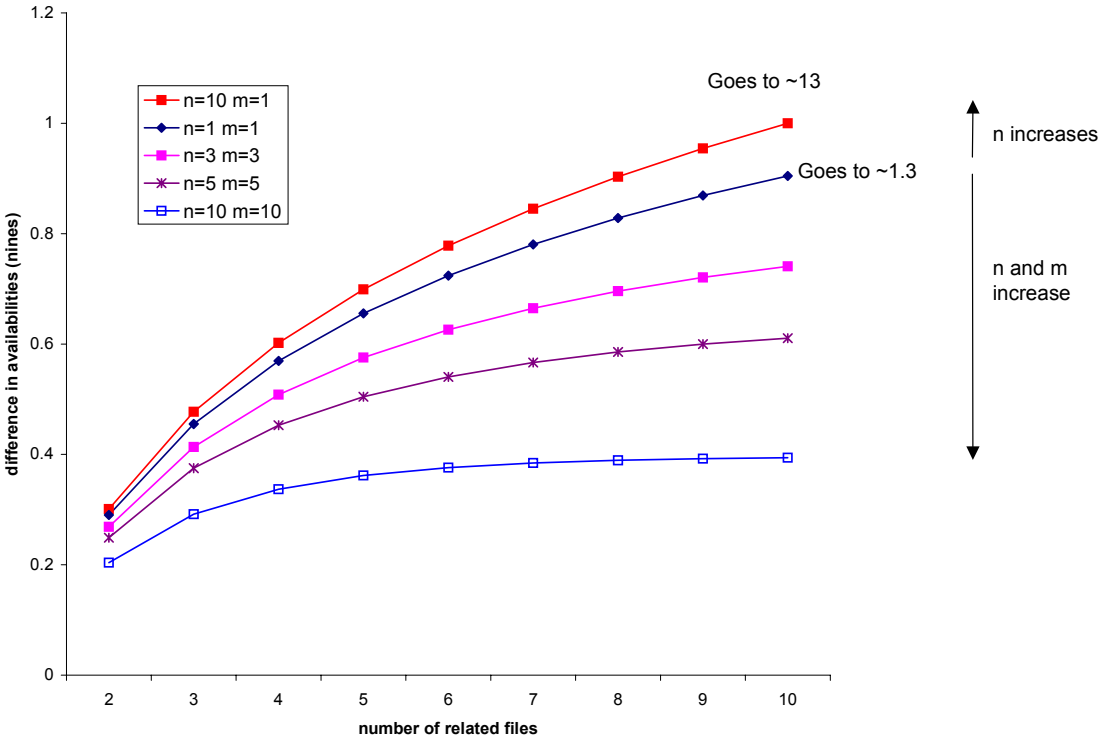


Figure 4-18. Effect of Having Related Files (Assuming Independent Failures)

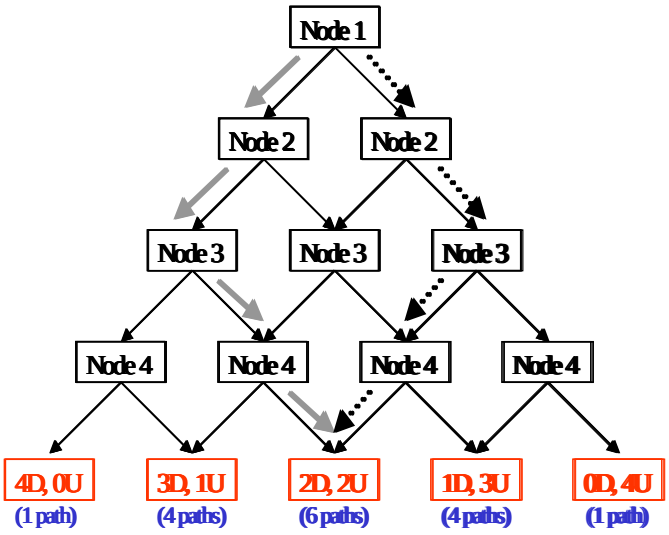


Figure 4-19. Two file example for the scheme with n=2 and m=1

The most prominent difference of Figure 4-20 from Figure 4-18 is that the difference (i.e., the value shown on the y-axis) for the schemes with $n=m$ is lower. The reason is, when correlation is introduced, nodes tend to fail simultaneously. Thus, when the files are stored on different sets of nodes they benefit from correlation. In the extreme case when correlation is 1, the difference is zero. For schemes with $m<n$, the difference is still important. Also, it is worthwhile to note that here the actual availabilities are lower. Therefore, a 1 nine difference in this figure is more crucial compared to Figure 4-18. For example, con-

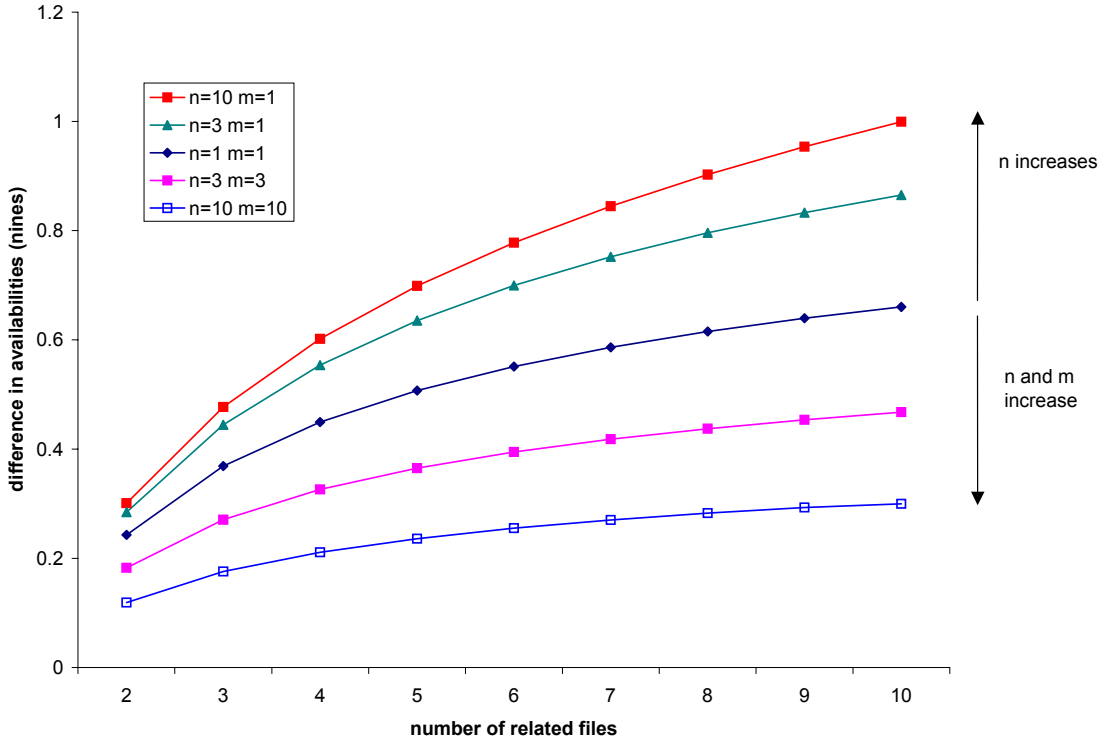


Figure 4-20. Effect of Having Related Files (Failures are Correlated)

sider the scheme with n=10 m=1, for the case when there are 10 related files. When the failures are independent the difference is (13.01-12.01), but, when the correlation is 0.25, the difference is (4.89-3.89). In the former, the difference accounts for ~0.0000283824 seconds unavailability per year, whereas in the latter the difference is ~1.02 hours unavailability per year. Based on this analysis, it is more important to take care with regard to related files when the failures are correlated.

In doing file placement, the system designer should consider relations between files. The importance becomes more substantial when the number of related files increases.

5 SECURITY IN PASIS

Security is one of the essential parameters in designing survivable storage systems. Choices of different *p*, *m*, and *n*s for a (*p, m, n*) scheme can affect the security of the scheme. A lower *p* value, for example, indicates that an adversary must acquire a smaller number of shares to obtain useful information. We need a model for storage security that can be used to make trade-off decisions. In this section we describe a modeling framework designed to analyze and quantify storage system security. Our work is loosely based on Schneier's attack trees [Schneier00] and the fault-tree analysis models [Vesely81].

5.1 The Attack Tree Methodology

Computer system security must be considered in the context of the system architecture and the operational environment. To capture all that, we need a model that represents the make up of the system, and in the mean time abstracts away the unnecessarily low-level details. In addition, the model must allow parametric analysis in order to isolate the impact of various components and different factors.

Fault trees [Vesely81] are frequently used for reliability analysis of critical systems. The fault-tree methodology is well understood and is one for which exists well-known synthesis and analysis methods [Fault81]. However, because fault trees are designed specifically for non-malicious fault types, they cannot easily capture notions of malicious behavior (e.g., timing attacks). Markov-chain models are an alternative that is flexible enough to model nearly any dynamic systems. However, the construction of a Markov model for any but the simplest system is tedious and error prone.

We propose the use of attack trees to model malicious attacks and evaluate computer system security. An attack tree is similar to fault trees in that the tree comprises a set of modeling constructs and composition mechanisms, which collectively models the various possibilities that the root event can happen. Attack trees can capture dynamic aspects of the system and the notion of malicious adversaries. The concept of attack trees was first proposed by Schneier [Schneier00].

Two types of fundamental constructs exist in an attack tree: *Events* and *Gates*.

Events: An event is a node in an attack tree that represents a security-relevant event (e.g., capture of a user password).

Gates: A gate is a composition mechanism that links a parent to the children nodes. A gate operation represents how a parent event can occur given the occurrence of its children events.

The property associated with each event in our attack tree methodology is *effort*, that is, the effort an adversary must expend to cause the event to occur. We use the metric of time to quantify effort. Because different adversaries may have different resources, which will result in varying amounts of time for the same event, implicitly associated with each attack tree is an adversary profile that captures, for example, the resources the adversary has at his or her disposal.

As with fault trees, the root node in an attack tree represents a particular attack goal and the entire tree represents the amount of time a particular adversary must invest to achieve the top-level event.

The various gates types are defined and demonstrated in the following sections.

5.1.1 Gates

A gate in an attack tree represents a composition operation between a parent node and its children. The main gate types are *OR*, *AND-Sequential*, *AND-Parallel*, and *M-N* gates. All the gates have a parent event and two or more children events.

OR Gate: The parent event of an OR gate (see Figure 5-1) occurs if any of the children events have occurred. The effort involved in causing the parent event is the minimum of the effort with all the children events

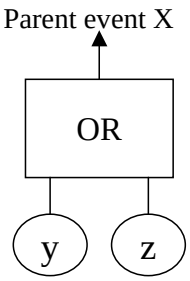


Fig. 5-1. An OR gate

In an OR gate, the timing sequence of the children events has no effect on the parent event—as long as any of the children events occur, the parent event is assumed triggered. The effort associated with the parent node X, E(x), is calculated as follows.

$$E\ (\mathrm{X}) = \min\ (E(\mathrm{y}), E(\mathrm{z}))$$

AND Gate: In an AND gate, the parent event occurs if all of the children events have occurred, and if they do not occur in parallel to each other.

In Figure 5-2, the event *X* occurs when *y* and *z* have both occurred. The timing sequence of *y* and *z* relative to each other is not important. The essential requirement is that *y* and *z* do not occur at the same time. The effort associated with the parent node *X*, E(x), is calculated as follows.

$$E\ (\mathrm{X}) = \mathrm{sum}\ (E(\mathrm{y}), E(\mathrm{z}))$$

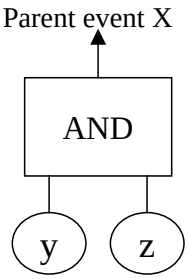


Fig. 5-2. An AND gate

AND-Sequential Gate: The parent event occurs if all of the children events have occurred, and if they occurred in the particular sequential order specified by the model.

The timing sequence of the children events in an AND-Sequential gate is important. In Figure 5-3, the parent event *X* only occurs when event *y* occurs before event *z*. Note that the temporal notion captured in this gate operation does not exist in the classic fault tree methodology. The effort associated with the parent node *X*, E(x), is calculated as follows.

$$E\ (\mathrm{X}) = \mathrm{sum}\ (E(\mathrm{y}), E(\mathrm{z}))$$

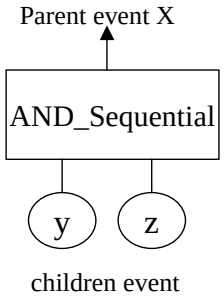


Figure 5-3.
AND_Sequential gate

Note that while the composition function for an AND gate is the same as that of an AND-Sequential gate, the two are conceptually different, and they can be used to model different scenarios. Consider the example of off-line dictionary attacks on static passwords. An adversary must first obtain the encrypted password file in order to mount the dictionary attack. An AND-Sequential gate can reflect the ordering of the two events. Consider yet a different example where a piece of data is divided into two pieces and stored on two different nodes,

AND-Parallel Gate: The parent event of an AND-Parallel gate occurs if all of the children events occur, and the children events occur in parallel. For an AND-Parallel gate, the effort associated with the parent node *X* is the maximum effort of all of the children nodes.

$$E\ (\mathrm{X}) = \mathrm{maximum}\ (E(\mathrm{y}), E(\mathrm{z}))$$

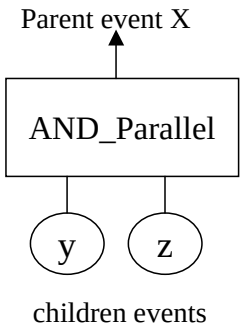


Figure 5-4.
AND_Parallel gate

In Figure 5-4, event *y* and *z* take place in parallel. *X* occurs when both *y* and *z* have occurred, and by definition, the effort involved in causing event *X* amounts to the maximum effort of *y* and *z*.

M-N: For an M-N gate, the parent event occurs if any *m* of the *n* children events have occurred. This is a special type of AND gate. The effort associated with the parent event is the minimum of the effort for any children subset of size *m*.

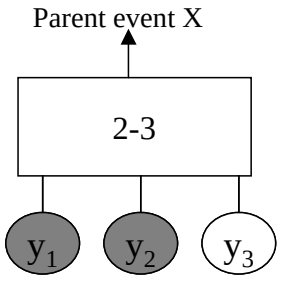


Figure 5-5. A 2_3 gate

$$E(\mathrm{X}) = \text{minimum}\left(\sum (E(y_{ij})\right), \text{ where } 1 < i < \mathrm{n}, \text{ and } 1 \le j \le m$$

Figure 5-5 shows a 2-3 gate where $E(\mathrm{x}) = E(y_1) + E(y_2)$

M-N-Parallel: The parent event occurs if any *m* of the *n* children events have occurred and they occur in parallel. Shown in Figure 5-6, the effort associated with the parent node *X* in an M-N-Parallel gate can be calculated as follows.

$$E(X) = \text{minimum}\,(\text{maximum}\,(E(y_{ij})),$$

where $1 \le i \le n$, and $1 \le j \le m$

Leaf nodes in an attack tree are events for which we can quantify the attack time. For example, a leaf node may be used to "break a 256-bit RSA key," which can be accomplished within two hours on a 1GHZ Pentium 4 machine.

5.2 How it Will be Used

The attack tree methodology can be used for the following purposes:

Identify the minimum effort required to breach the system security (i.e., achieve the attack goal at the root node).

Identify the most vulnerable points in the system, i.e. the components that are easier to break than others.

Compare two different systems in terms of the minimum attack effort.

An attack tree is constructed in a top-down fashion: a high-level security property is decomposed into children events, and each of the children events is further decomposed until we reach an event for which we have a reasonably accurate picture of the effort involved in instigating that particular event, at which point we deem this event a leaf node and do not decompose any further. The construction of the tree completes once every branch has been pursued to its leaf node.

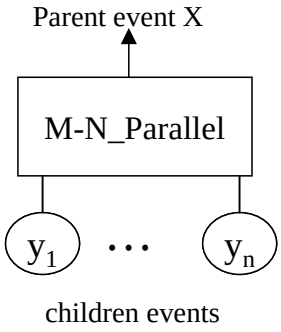


Figure 5-6.
M_N_Parallel Gate

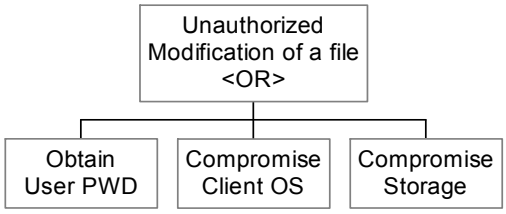


Figure 5-7. The Top level decomposition

The evaluation phase starts at a bottom-up fashion. The effort in achieving a parent attack event can be computed from the efforts of its children nodes as defined by the various gate types. We start from the leaf nodes and work our way up to the top-level event, at which point the evaluation completes with a final outcome—the effort an attacker must expend to achieve the top-level event.

To see how it works, consider the security property "no unauthorized modification of a user file" in a UNIX-based NFS file system. This high-level property can be represented in the attack tree framework as a top-level node — "unauthorized modification of a file," and this node can be decomposed into three attack events, *obtain user password*, *compromise client OS*, and *compromise storage,* in an OR relation. This decomposition is shown in Figure 5-7.

The event, obtain user password, can be further decomposed as shown in Figure 5-8. The shaded nodes represent leaf nodes. For the purpose of discussion, we deem *social engineering* a leaf node. The problem of quantifying social engineering effort to obtain user password is beyond the scope of the attack tree methodology. We simply assume that it can be done and we can attach an effort number with this node.

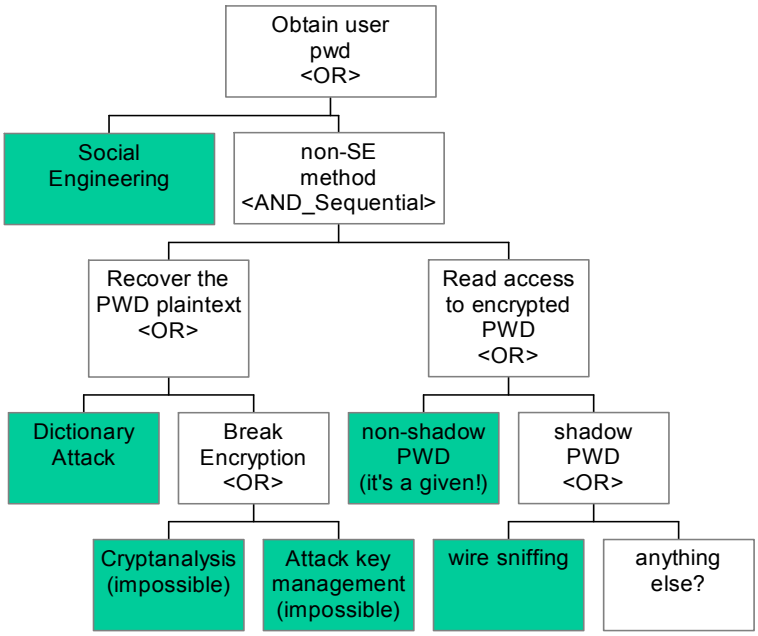


Figure 5-8. Decomposition of password compromise

In terms of the non-social-engineering methods, one must first obtain read access to the encrypted passwords and then launch whatever attacks are necessary to discover the plaintext of the password. The two operations, *gaining read access* and *recover the password*, must be performed in sequence; hence an AND-Sequential gate is used.

We assume the password files are stored on the server, encrypted with a 25-round DES algorithm. The first DES round uses 64 0-bits as input and the user password as the key. The output of the first round is then fed into the subsequent rounds, salted and with the same key. The result of the 25th round is stored in the /etc/passwd file. Reversing this encryption process is virtually impossible. Since the user password is used as the key to the DES encryption, one cannot exploit any weakness in the key management process. Cryptanalysis against such a DES algorithm is not known to be effective.

In the case that a shadow password file is used, normal users do not have read access to the encrypted password. To obtain the encrypted password, an attacker may perform wire sniffing (tap into communications between the client and the server). The successful rate of straightforward wire sniffing attacks can be measured. We therefore stop further decomposing.

The same process can be used to decompose the events, compromise the client OS and compromise the storage node. Once we have all the leaf nodes and their associated numbers, we can compute the effort associated with the top-level attack event: unauthorized modification of a user file.

Of course, construction of an attack tree for a real system is not as simple as it seems in the example. A thorough understanding of the system and its various underlying components is a prerequisite, and that alone is not easy to obtain if the system in question is complex (and all real systems by definition are.) In addition, it is not always easy to evaluate the leaf nodes. For this methodology to work, there must be standard metrics upon which the basic attacks can be measured.

6 PERFORMANCE IN PASIS

Digital information is a critical resource, creating a need for distributed storage systems that provide sufficient data availability and data security in the face of failures and malicious attacks, while offering competitive performance. Many research groups [Wylie01, AndersonD00, eVault, Farsite, FreeHaven, Intermemory] are now exploring the design and implementation of such *survivable storage systems*. These systems built on mature technologies from decentralized storage systems [AndersonD00, Castro00, Gibson98, Lee96, Liskov91] and share the same high-level architectures.

One key to maximizing survivable storage performance is wisely selecting the *data distribution scheme.* A data distribution scheme consists of a specific algorithm for data encoding & partitioning and a set of values for its parameters. There are many algorithms applicable to survivable storage, including encryption, replication, striping, erasure-resilient coding, secret sharing, and various combinations of these. Each algorithm has one or more tunable parameters. The result is a large toolbox of possible schemes, each offering different levels of performance (throughput), availability (probability that data can be accessed), and security (effort required to compromise the confidentiality or integrity of stored data). For example, replication provides availability at a high cost in network bandwidth and storage space, whereas short secret sharing provides availability and security at lower storage and bandwidth cost but higher CPU utilization.

There is a wide variety of reasonable data distribution schemes, offering very different levels of availability and security guarantees with different performance tradeoffs. As a result, no single data distribution scheme is right for all systems. Instead, the right choice for any particular system depends on an array of factors, including desired levels of availability and security, expected workload, and system component characteristics such as server performance and client processing power. Unfortunately, most system designs appear to involve an ad hoc choice, often resulting in a substantial performance loss due to missed opportunities and over-engineering.

Wylie et al. [Wylie01] propose a better approach to selecting a data distribution scheme. This method involves:

- Enumerating possible data distribution schemes,
- Modeling the consequences of each scheme, and
- Identifying the best-performing scheme for any given set of availability and security requirements.

These results are plotted in a 3-D graph with security, availability and performance axes (Figure 1-1). Specifically, the surface represents the performance of the best-performing scheme that provides at least the corresponding levels of availability and security. This method serves two functions: (1) it enables system architects to make informed trade-offs among security, availability, and performance, and (2) it identifies the best-performing scheme for each point in the trade-off space.

A class of algorithms, known as threshold schemes, is used by some proposed survivable storage systems. PASIS [Wylie01, PASIS] is a distributed storage system targeting storage in WANs and LANs. Threshold schemes break a file into n shares, any m of which are sufficient to recreate the original file. In PASIS, each of these shares is written out to a different server. Although threshold schemes increase the availability [as defined in Siewiorek92 and Gray91] and security of data, they hurt performance in two ways. First, the encode-decode algorithms can be CPU-intensive. This can be addressed by making the algorithms more efficient, using hardware random-number generators, and by using faster CPUs. Second, communicating with multiple servers and having to wait for multiple responses. In this paper, we concentrate on improving the performance of share-retrieval in a survivable storage system. We explore and evaluate two methods:

- *Over-requesting:* this involves requesting more than the required *m*, and using the first *m* that come back.
- *Intelligent Server Selection:* this involves requesting shares from those servers that are expected to be the fastest. Instead of randomly selecting which servers to request from, clients collect statistics and use them when selecting amongst the *n* servers.

To show the value of these policies, we created and verified a model of response times for survivable storage in a WAN. We use this model to explore various scenarios and understand the costs and benefits associated with the two methods.

While any one client can use either of these methods to improve its performance without causing widespread performance repercussions, global implications must be considered when many clients are using the system. In section 6.4.2, we explore local versus global decision-making and discuss avoiding the "tragedy of the commons" [Hardin68, Turner93].

For the 3-D graph approach to be effective for making design decisions, we need, among other things, to have an accurate understanding of how well each scheme will perform. It has been shown that in order to make good decisions, we need to be able to predict the performance of network share-retrieval to within a factor of two, if the rest of the system does not change [Wylie01]. We develop two models: *static* and *online*. The static model makes predictions of read transaction times given an average-case model of servers. Here, we show that the performance predictions can hold true to within 28%, much less than the factor of two necessary to make good design decisions.

The online model uses recent history to dynamically alter its model of the servers, thus yielding more accurate predictions and that can be used in a running system. We show that the online model has a mean percent error of less than 26% and a 50th percentile error of less than 19%.

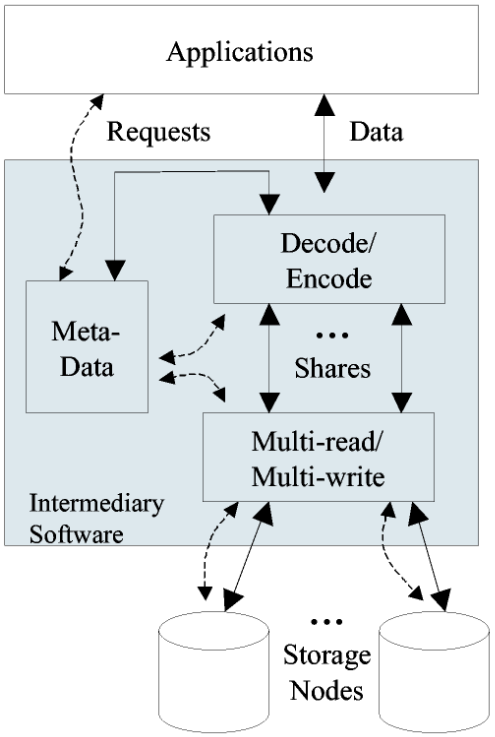


Figure 6-1: High-level architecture of most decentralized storage systems.

The work in this chapter may be found published as "On Modeling and Improving the Performance of Threshold Scheme-Based Distributed Storage Systems" [Pandurangan02].

6.1 Background

6.1.1 Survivable Storage Systems

Survivable systems operate from the fundamental design thesis that no individual service, node, or user can be fully trusted; having some compromised entities must be viewed as the common case rather than the exception. Survivable storage systems encode and distribute data across independent storage nodes, entrusting the data's persistence to sets of nodes rather than to individual nodes. If confidentiality is required, unencoded data should not be stored directly on storage nodes; otherwise, compromising a single storage node enables an attacker to bypass access-control policies.

Prior work in cluster storage systems [AndersonD00, Castro00, Gibson98, Lee96, Liskov91] provides much insight into how to efficiently decentralize storage services while providing a single, unified view to applications. Figure 6-1 illustrates the generic decentralized storage system architecture.

The intermediary software is responsible for presenting a seamless translation between the single unified storage system seen by the clients and the decentralized reality. This intermediary software may operate on each client independently, on intermediary nodes, or on certain specialized servers.

In most cluster-based storage systems, the data and some form of redundancy information are striped across storage nodes. Survivable storage systems differ from decentralized storage systems mainly in the encoding mechanism used (i.e., the data distribution scheme). A well-chosen data distribution scheme enables the storage system to survive expected numbers of failures and compromises of storage nodes.

6.1.2 Threshold schemes

There is a wide array of data distribution algorithms, including encryption, replication, striping, erasure-resilient coding, information dispersal, and secret sharing. Threshold schemes, characterized by three parameters *n*, *m*, and *p* (where $n \geq m \geq p > 0$), represent a large set of these algorithms. Data is divided into *n* shares, any *m* of which are sufficient to fully recover the data. The *n* shares formed during encoding can be stored onto different storage nodes. This increases availability, since the system can function in spite of up to *n-m* server failures. Additionally, for an adversary to steal the data in its entirety, he or she must compromise at least *m* storage nodes. The *p* value conveys information-theoretic security; stealing less than *p* shares provably gives an adversary absolutely no information about the stored data.

Commonly-known Name	Threshold Scheme ***n-m-p*** values	Total size of data stored (one replica is 1)
Replication	*n*-1-1	*n*
Decimation (Striping)	*n-n*-1	1
Splitting (XORing)	*n-n-n*	*n*
Information Dispersal	*n-m*-1	1
Secret Sharing	*n-m-m*	*n*
Ramp Schemes	*n-m-p*	*n/(m-(p-1))*

Table 6-1: Commonly-used encoding schemes, their *n-m-p* representations, and the total size of the stored data

Table 6-1 lists some *n-m-p* threshold schemes that have more familiar names. Perhaps the simplest example is *n*-way replication, which is an *n*-1-1 threshold scheme. That is, out of the *n* replicas that are stored, any single replica provides the original data (*m*=1), and each replica reveals information about the encoded data (*p*=1). Another simple example is decimation (or striping, as in disk arrays), wherein a large block of data is partitioned into *n* sub-blocks, each containing 1/*n* of the data (so, *p*=1 and *m*=*n*).

At the other end of the spectrum is "splitting", an *n-n-n* threshold scheme that consists of storing *n*-1 random values and one value that is the exclusive-or of the original value and those n-1 values; *p*=*m*=*n* for splitting, since all *n* shares are needed to recover the original data. Replication, decimation, and splitting schemes have a single tunable parameter, *n*, which affects their place in the trade-off space. With more CPU-intensive mathematics, the full range of *n-m-p* threshold schemes becomes available.

In this paper, our focus is on the performance of clients' read operations. Two main steps that constitute a read are: (1) fetching the shares from the servers, and (2) decoding them. We examine the potential for improvement in share retrieval times through the use of intelligent server selection and over-requesting.

6.1.3 An Overview of the PASIS Architecture

PASIS is a prototype survivable storage system, PASIS [Wylie01, Wylie00]. It fits the architecture illustrated in Figure 6-1, with the intermediary software executing on the client machines. The encode/decode and multi-read/multi-write aspects of the architecture are implemented in separate libraries. PASIS allows administrators to select from many data distribution algorithms and to specify a wide range of parameters enabling the exploration of many schemes within the context of a single survivable storage system.

The encode/decode library supports all of the basic schemes described in Section 6.1.2. The library interface exports two main functions: (1) a decode call that takes a scheme description and a set of shares as input and returns the original data as output, and (2) an encode call that does the reverse.

The multi-read/multi-write library manages parallel communication with a set of storage nodes. The library interface exports two main functions: (1) a fetch call that takes a set of *n* share descriptions as well as a number of shares required (*m*), and (2) a store call that does the same for writes. Each share description includes a storage node description and a name for the share within that storage node.

When implemented in PASIS, the server selection algorithms (algorithms that clients use to select which servers to request from) described in this paper will fit in the multi-read / multi-write library. Before each fetch operation is conducted, the algorithm will select a group of servers to request from, and will record the response times from each of these servers so that future requests can make use of these historical data.

The multi-read/multi-write library currently interfaces with a variety of common off the-shelf software (COTS) backend protocols, including NFS storage nodes, CIFS (Microsoft's Network Neighborhood) storage nodes, and FTP storage nodes. The use of COTS software is an important practical design consideration that allows survivable storage systems to enhance storage node heterogeneity, which in turn enhances security; for example, the likelihood is low of finding a single attack that can compromise a Sun NFS server, a Network Appliance NFS server, and a SNAP NFS server. Without this feature, the effort to break into the second through *n*-th storage nodes could be negligible. Since no monitoring or instrumentation code can be added to the COTS components on backend servers, PASIS and similar systems are limited to collecting only those performance data that can be observed from the clients.

6.1.4 Over-Requesting and Server selection

When reading data that has been stored using threshold schemes, a client need only fetch *m* of the *n* shares; *m* shares are sufficient to fully reconstruct the original data and fetching the minimum places the least load on the servers and the network. There are three reasons fetching only *m* may be sub-optimal:

1. As observed through measurements, server response times often have high variance both across servers and for a given server.
2. One or more of the storage nodes might not be available [Powell00, Bakkaloglu02].

Some of the retrieved shares may have been corrupted.

In case 1, one slow server of the *m* will slow down the entire read operation, since all *m* shares need to be present before the decode operation can complete. In case 2, the system would time out and have to re-request additional share(s). In case 3, upon discovering that a share is corrupt, the client would need to fetch another share.

Over-requesting can help to mitigate these effects. By requesting more than the minimum *m* shares, it becomes more likely that a client receives at least *m* shares quickly. Therefore, requesting *r* shares (where $n \geq r > m$) is defined as over-requesting. Since we can have multiple levels of over-requesting, *r* is defined as the over-requesting level.

Intelligent server selection is the process by which clients choose those servers that they believe are most likely to have the fastest response times. Clients collect historical performance data about servers and use this data to reduce the likelihood of picking slow servers. Using intelligent server selection in conjunction with over-requesting (i.e. requesting shares from those *r* servers with the fastest expected response time) may produce the most effective *server selection algorithm*. We refer to an over-requesting algorithm that does not utilize any historical data as *random over-requesting*; when implementing this algorithm, clients randomly select *r* out of *n* servers from which to request shares for each read operation.

6.2 Description of Models

The share retrieval performance model predicts end-to-end latency for a data-retrieval operation, defined as a read call made by the client application to the intermediary software. The intermediary software translates this individual read into one or more read requests sent out to servers. The operation completes when the intermediary software receives enough shares (m) to reconstruct the data requested by the client. The latency predicted is the time between requests being sent out for r shares and the client receiving m of them. This section gives background required to understand the model, presents the generic static and online models and discusses the differences between their use in the LAN and WAN.

A simulator was constructed to explore the global effect on servers' response times when multiple clients use a particular server selection policy. Section 6.3 describes the simulator, and Section 6.4 uses it to explore the global performance implications of various server selection algorithms.

6.2.1 Background

The goal of the performance model is to be able to predict the performance of future read operations using data previously collected about the relevant servers. Complicating this task is that fact that we wish to accomplish this based solely on observations made by the client.

The response time observed by the client has two components: server latency and network transfer time. Figure 6-2 illustrates a read operation. Server latency is the delay between the client sending a request and the client beginning to receive a response. The network transfer time is the period between the client beginning to receive a response and the response being received in its entirety.

Server Latency

The model can use information about the observed latency to infer the various phases of data transfer and to ensure that the network resources are appropriately scaled so it does not predict a transfer rate of more than the maximum available bandwidth.

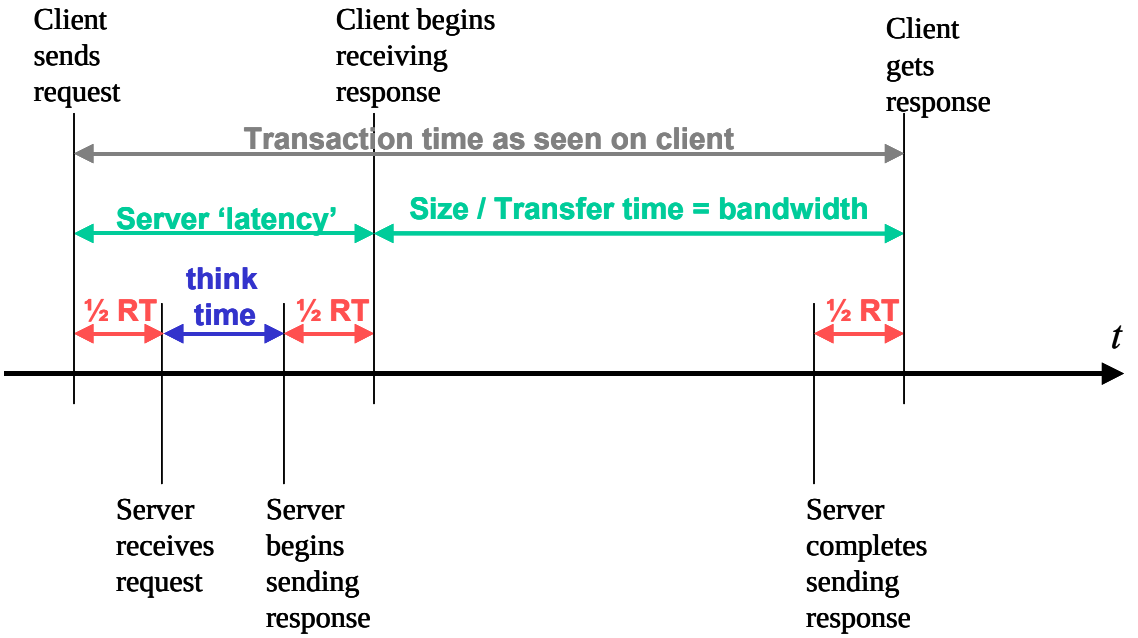


Figure 6-2: Anatomy of a read operation. The client is only able to observe the overall transaction time. The duration of the server latency may be estimated by the period between when the client sends the request and when the server begins responding. The think time can be inferred by pinging the server and subtracting the round trip time from the latency.

To make predictions more accurate, the server latency is broken into two components:

The round trip time – The round trip time of the network can be obtained by pinging the server, utilizing information from TCP's round trip time estimator [Karn87], or using recent requests of different size to solve for the round trip time [Kim01]. In most cases, the client will experience one round trip time in addition to the think and transfer times.

The think (service) time – The think time is composed of two separate delays: the queuing delay and the time to process this specific request. We separate the think time from the round trip time because the latter is more easily observed, and because they will vary differently. If the round trip time is known, think time may be computed by subtracting one round trip time from the measured server latency. One could measure the server latency by recording the time when data first begins arriving at the client, though the software organization of most systems obfuscates this time.

In a WAN, the network transfer and round trip times generally dominate the think time. Additionally, a well-connected client's local connection is usually not saturated in a WAN. Generally, in LANs, the round trip time is more consistent, server latency dominates network time, and the client's local connection can sometimes be the bottleneck. Thus, predicting the server latency independently may be important in getting better predictions of performance in a LAN.

Correction for over-predicting bandwidth

In some cases, the model may predict a bandwidth from a set of servers that exceeds the maximum possible bandwidth of the local connection. When this occurs, the model must cap the predicted bandwidth at the maximum actual bandwidth of the local connection. An example follows: When downloading large shares from four servers over a 56K modem, although each server's throughput is predicted to be less than 56Kbit/sec, their collective throughput cannot be forecast to be larger than this value. Thus, the model reports a throughput rate of 56Kbit/sec. The local connection's bandwidth is supplied as a parameter to the model. In a case where the latency no longer dominates the network transfer time, this simple strategy is not sufficient. Figure 6-3 illustrates this point. If both servers one and two use 80% of the maximum bandwidth servicing their request but have different latencies, there will three different phases of network utilization. Simply predicting the maximum of the two observed times does not work, as illustrated in Figure 6-3A. Doing that would result in a prediction where nearly 200% of the maximum bandwidth is used for a period of time. The solution is to scale the regions where the network utilization would have to be more than the client's local link bandwidth to 100% and extend the length of the operation. This is illustrated in Figure 6-3B.

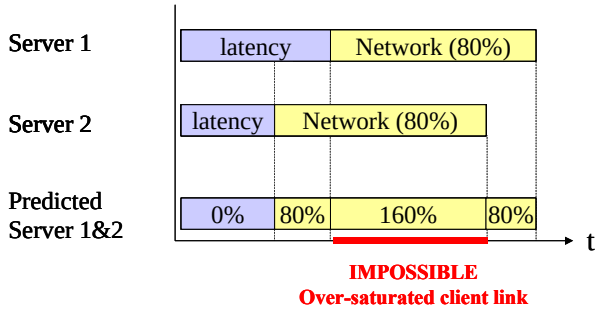


Figure 6-3A: This illustrates the downfall of predicting combined server time with individual measurements when latency is small compared to the network time.

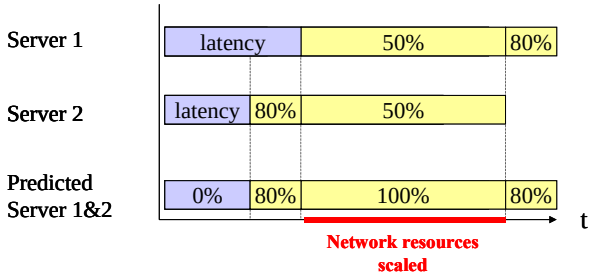


Figure 6-3B: The LAN model corrects this situation by determining all the different transfer phases and scaling network resources appropriately.

Figure 6-3: Illustration of predicting joint request time from multiple servers

6.2.2 The Model

A model was constructed in Matlab to evaluate the efficacy of various algorithms in predicting response times of servers in a WAN. The model can operate in either a static or an online mode. For the online mode, the model can use historical server response times either from a trace or can generate them based on a probability distribution.

During validation, the model's predictions are compared against values from the trace to determine the algorithms' efficacies. This procedure is described in further detail in Section 6.3.

Static Mode

When a survivable storage system is initially provisioned, various possible encoding schemes must be considered. In order to choose the correct scheme off-line, a predicted performance must be available for each scheme. The model operates in static mode to generate such data.

When the model operates in static mode, it accepts as an input a summarized statistic (for example, a random variable) for each server. It then uses these inputs to generate the predicted response time for an operation with given *n*, *m*, and *r* values. The response time for such an operation is determined by making a prediction for each of the *r* servers (using the summarized statistics) and selecting the *m*-th fastest completion time. Since the variance of the servers' response times will affect the response time of an <*n*,*r*,*m*> retrieval operation, numerous simulated operations are made and the mean operation response time is reported for provisioning purposes. The static mode can make predictions for a client using the two server selection algorithms discussed in section 6.1.4: intelligent server selection (with and without over-requesting) and random over-requesting.

In the future, the static mode could be expanded to employ a flip-flop-based filter [Kim01], such that multiple static predictions could be stored and chosen between depending upon the file size or the time of day during which we expect the operations to be conducted.

Online Mode

The model will operate in the online mode in order to optimize share-retrieval times in a running client. In lieu of using a summarized statistic for each server, the online model employs various predictive algorithms that use historically observed response times to make predictions for server response times. This allows the client to adapt to changing trends in server response times. In our experiments, the model accepts historical data for each server through a trace. When predicting individual response times, the model feeds previous (historical) readings to a selected predictive algorithm. The predictive algorithms used are discussed in the next section. In the evaluation phase, we tested the algorithms and found that the stability filter proposed by Kim et al [Kim01] was most effective.

Because they are in running clients, the execution time of these algorithms is an important factor in determining their usability. All the algorithms included here run in O(*previous requests*) or better per request. In most cases, however, this performance penalty applies only for the first request made to a set of servers, because the result of that prediction, if cached, can be used to make predictions about future requests. If caching is performed, future requests will have a constant execution time.

Online Predictive Algorithms

The following predictive algorithms were evaluated:

Mean of previous *k* samples – This algorithm uses the mean of the previous *k* samples as the predicted value of the next request. The value of *k* is supplied as a parameter that is unchanging and used for all servers.

Median of previous *k* samples – This algorithm uses the median of the previous *k* samples as the predicted value of the next request. As above, the value of *k* is supplied as a static parameter that is used for all

servers. This algorithm is more stable than the above for larger *k* values, because it is not as susceptible to being skewed by outlying points.

- Exponentially-weighted moving average (EWMA) filter – Such a filter can be mathematically represented as follows,

$$E_{t+1} = \alpha E_t + (1-\alpha) O_t \tag{6.1}$$

where E_{t+1} is the prediction for time t+1, E_t was the prediction for time t, O_t was the value observed at time t, and α is the gain. The gain determines the agility of the filter. Low α values result in an agile filter, which treats recent observations with great weight, whereas high values of α result in a stable filter. This generic class of filters is used in many control systems, including the round trip time estimator in TCP [Jacobson88]. (TCP's round trip time estimator is quite stable and uses an α value of 0.875).

Stability filter – The stability filter [Kim01] uses a generic EWMA filter. The problem with a simple EWMA filter is that the α value is fixed, making it difficult to create one filter that is good in a variety of situations. In order to correct this, filters have been proposed that allow us to dynamically change the value of α. One such filter is the stability filter. It changes α proportionally to the variance in the observed values. This has the effect of causing the filter to become stable in cases where consecutive observations vary significantly. Thus, the filter becomes less susceptible to random and transient changes in network performance. However, if there has actually been a large shift in the performance of the system, the observations will soon seem to settle, and thus the filter will adjust appropriately. The stability filter is defined by the following equation:

$$(1) \qquad U_t = \beta U_{t-1} + (1-\beta)\left|O_t - O_{t-1}\right|$$

$$(2) \qquad U_{\max} = \max\left[U_{t-9}, U_{t-8}, \cdots, U_t\right] \tag{6.2}$$

$$(3) \qquad \alpha_t = \frac{U_t}{U_{\max}}$$

Error filter – This filter, also proposed by Kim and Noble [Kim01], is similar to the stability filter. However, the error filter adjusts α based on the predictive power of previous estimates. Instead of becoming less agile with increased variance in the observations, the error filter becomes less agile when the predictions are close to observed values. This results in a filter that is able to quickly adapt to large changes in values, but less susceptible to small performance changes. The error filter is defined by the following equation:

$$(1) \qquad \Delta_t = \lambda \Delta_{t-1} + (1-\lambda)\left|O_t - E_{t-1}\right|$$

$$(2) \qquad \Delta_{\max} = \max\left[\Delta_{t-9}, \Delta_{t-8}, \cdots, \Delta_t\right] \tag{6.3}$$

$$(3) \qquad \alpha_t = 1 - \frac{\Delta_t}{\Delta_{\max}}$$

6.2.3 Multiple clients

The model above predicts the performance of operations from a single client. It can show the improvement in performance expected when one client alone implements over-requesting and intelligent server selection. A simulator was designed to explore the effects of having multiple clients all implementing a selection strategy concurrently. It simulates a system where a number of clients make requests for shares from a group of *n* servers. Each server and client operates as a separate thread. Each client is a closed system. It makes a request for a shared file and, upon the receipt of the first *m* shares, waits some time (the client think time) before making a subsequent request. Each server has a finite queue of requests and a processing time for each request. When the client attempts to add a request to a server with a full queue, the request is dropped. From the client's perspective, the server will appear to have been unavailable. Both the think and processing times can be a random variable. In addition, responses from servers may be delayed by a network delay time. The network delay time can also be a random variable. The network delay experienced by all clients interacting with a specific server is derived from the same random variable (all clients have the same view of each server). The simulator collects various data such as server queue depths, average operation completion time, average response time of each server as seen from the clients, and total number of timeouts from each server. A more complete description of the *server* and *client* modules, along with the implementation of intelligent server selection follows.

Client

The client module models the behavior of each client in the system. For each request that must be made, a value for *r* and a corresponding list of servers are selected according to the server selection algorithm currently in use. The client then adds a request into the queue of each of the selected servers. The request contains the client's id, so the response can be routed to the correct client, and a sequence number, to ensure that responses that arrive after their operation has completed can be discarded. The client waits a specified timeout interval for *m* responses. As each response is received, the response time of the relevant server is recorded. These recorded response times are used by some server selection algorithms to choose the set of servers from which to make subsequent requests. As soon as the *m*-th response is received, the operation completion time is recorded, and the client sleeps for its think time before sending out a new request. If *m* responses are not received before the timeout interval expires, the client will time out and send out new requests to a set of servers suggested by the server selection algorithm.

Server

The server module models the behavior of each server in the system. Each server has its own request queue, which has a finite length. If a client attempts to add a request to a full queue, the request is dropped, making the server unavailable from the client's perspective. As long as there are jobs present in the queue, the server

1. Takes the first request off the queue
2. Pauses for the appropriate processing time
3. Places a response in the client's queue and returns to step 1. The responses arrive after a network delay, which is derived from a random variable.

When no jobs are present in the queue, the server waits for a new job to arrive.

The server periodically records the length of its queues to aid in analysis.

Intelligent Server Selection

This simulator can implement intelligent server selection using any of the predictive algorithms specified in section 6.2.2. Each client keeps track of a statistic for each server it visits. Before making a request

from a group of servers, the client looks at the predicted response times for each of the servers. The client can then intelligently request shares from the *r* servers that are predicted to be the fastest. After a response is received from a server, the client updates the appropriate server's statistics. Note that each client collects statistics on the servers on its own, and thus makes decisions independently of other clients.

Upon analysis (5.2), it was discovered that always selecting only the *r* fastest servers is usually not a globally (or even locally) optimal behavior. Periodically, the client must sample the (*n-r*) other servers in order to update its statistics of those servers. Otherwise, some servers may never be visited, and clients would not capitalize on any changes in those servers' performance. The client randomly selects one of the "un-intelligent" servers for over-requesting every other or every third request.

6.3 Validation

6.3.1 Predictive Performance Model

In order to validate the predictive model, we collected performance data from a group of 106 servers over a period of 8 months. Every 10 minutes, each server's response time to a request for the home page was recorded. (The response time is defined as the time period between the application sending out the request for the file and receiving the complete response, as in Figure 6-2.) This results in over 30,000 data points for each server, or 3 million data points in all. In this section, we use this data to show that response times of HTTP servers in a WAN are generally predictable and that the model described in Section 6.2 is valid.

Trace collection

In order to collect traces, a group of 106 servers were selected. Some included heavily used, load-balanced sites such as www.cnn.com, less-heavily-used, non-load-balanced sites such as www.jeremyshafer.com, and international sites such as pmindia.nic.in. A full list of these servers is included in Appendices B-2 and B-3.

A Pentium III 550 MHz Dell-based machine was set up with Redhat 6.2-based Linux and connected to the Internet through a Carnegie Mellon University connection. A program was written to periodically poll the various servers and download the index file from each of them. Requests were staggered to prevent context-switching overhead from affecting the precision and accuracy of time measurements. The program starts a process for each server. The process polls each server, records the result, then sleeps for ten minutes. The algorithm is included in Figure 6-4.

```
for (ever)  {
 sleep (10 minutes);
 starttime=getTime();
 open file socket;
 send request;

 wait for HTTP OK response;
 operTime=getTime()-starttime;

 file.write (resp_size,time_of_day,
error_code, operTime);

}
```

Figure 6-4A

```
volatile unsigned long long get-
Time () {
 unsigned int low,high;
 unsigned long long l,h;
  __asm__(".byte 0x0f,0x31" :"=a"
(low), "=d" (high));
l=low;h=high;
  return (h<<32)+l;
}
```

Figure 6-4B

Figure 6-4: This is the algorithm used to collect response time data from each web server. For each server whose performance is monitored, a process executing this code is started. Operations are timed through the use of the performance counter on the Intel Pentium processor. The counters were accessed through assembly language function in Figure 4-B.

Server 1	Server 2	r- value
asia.cnn.com	www.cnn.com	0.557655
armedforces.nic.in	pmindia.nic.in	0.40282
msdn.microsoft.com	www.berkeley.edu	0.400202
www.cs.ucsd.edu	www.ece.cmu.edu	0.3659
www.ece.cmu.edu	www.microsoft.com	0.336335
www.cs.ucsd.edu	www.harvard.edu	0.331532
www.berkeley.edu	www.cabinet.gov.jm	0.320242
www.ece.cmu.edu	www.library.cmu.edu	0.304477
www.ece.cmu.edu	www.harvard.edu	0.302797

Table 6-2: Server pairs whose correlation r values are more than 0.3. See appendix A for the complete listing of all servers with an r-value of >0.1.

As mentioned earlier, we selected the index file (usually 'index.html') as the file to download. A consequence of this decision is that each server's index file has a different size, and further this size varies over time. For the purposes validating a model for a file system with a fixed block size, it would have been better to have the files downloaded from all servers be the same constant, fixed size. However, as a matter of practice, it is difficult to find (or place) identically-sized files on 106 different servers that are guaranteed to exist throughout the experiment. The fact that the file sizes vary over time presents another variable in the analysis, and as such, decreases the specificity of the conclusions drawn from this data set. In the subsequent subsections, we show that the model can predict the performance of servers within a factor of two even given this data set with the non-constant file sizes.

Analysis

Figure 6-5 shows a sample response time distribution for one server. Note that there are three main "clumps" of data points (at *a, b,* and *c* seconds). Since the file sizes varied over time, clumps *a* and *b* correspond mainly to two different ranges of file sizes: clump *a* was composed mostly of requests around 5KB; clump b was composed of mainly requests of 22KB. Clump *c* is caused by timeouts. In the cases where the server does not respond to connection requests correctly, a response time of 5 seconds is assigned. (The issue of predicting and modeling availability in a distributed storage system is discussed in [Bakkaloglu02].)

In order to come up with a simple method of modeling response times, we attempted to fit the data collected from each server to standard probability distributions. We tried for a fit with exponential, gamma, normal, and uniform distributions. In order to evaluate the effectiveness of each fit, we used a "goodness of fit" test using a chi-squared test and Scott's rule [Hansen]. None of the distributions was a good fit.

We examined the correlation between the response times of different pairs of servers. For each of approximately 5000 pairs (106 choose 2) of servers, we generated the statistical r value for all response time observations that occurred at the same time. (For more information on interpreting statistical r values, see Goldberg [98] and Bolosky [00]). A table of all statistical r values above 0.1 is included as Appendix A. Of all the pairs, there are 9 pairs with a statistical r value of more than 0.2. These are included in Table 6-2. As can be seen, the highest pairs include asia.cnn.com & www.cnn.com, and armedforces.nic.in & pmindia.nic.in. These pairs share a common route from our test machine all the way until the next-to-last network hop.

Static model

As mentioned in Section 6.2.2, the static model relies on few summary statistics to predict the response times of operations.

The first step to validating the static model is to look at the predictions of response times of requests from individual servers. First, the data was separated into a test set and a training set so that 80% of the data fell into the test set and 20% into the training set. This was done by assigning each data point into the

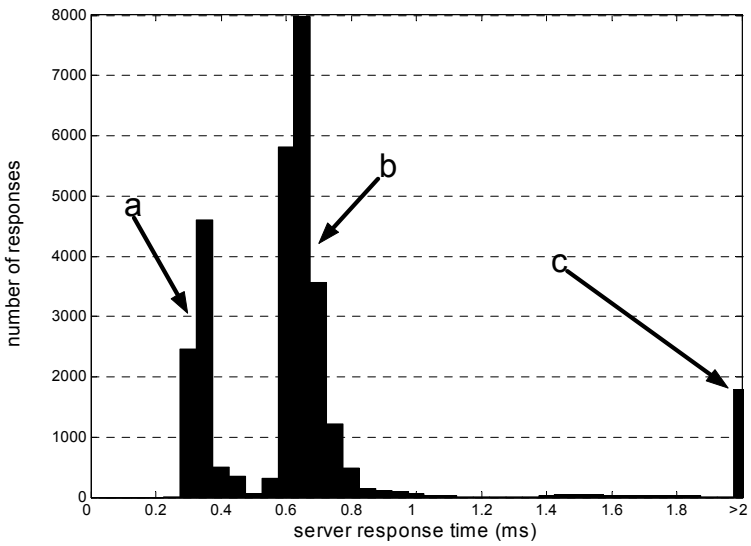


Figure 6-5: Histogram of server response times for a server. (www.bl.uk). The x-axis shows response time in seconds and the y-axis shows the number of requests that have that response time. All response times more than 2 seconds appear in the bar at 2-seconds.

training set with 0.2 probability. This is different than assigning every 5th point to the training set because there is no pattern in the distribution of the points, reducing the possibility of an unrepresentative set.

Using the training set, the mean and median values of throughput for each server were calculated. Using first the mean, then the median, as the predicted value for all requests in the test set, the absolute percent error for each of these requests was calculated for each of the 106 servers.

The absolute percent error for request *w* is defined as:

$$absPError_w \equiv \left| \frac{(predicted\_value_w - actual\_value_w)}{actual\_value_w} \right| \times 100\% \qquad (6.4)$$

In order to evaluate the performance of the algorithms across all servers, the following metrics are used to summarize the error the model shows for one server:

Median absolute error for one server over *q* requests:

$$median\lfloor absPError_0, absPError_1, absPError_2, \ldots, absPError_q \rfloor \qquad (6.5)$$

Mean absolute error for one server over *q* requests:

$$mean\lfloor absPError_0, absPError_1, absPError_2, \ldots, absPError_q \rfloor \qquad (6.6)$$

The results of our analysis show that the median statistic (with a mean absolute percent error of 32%) is a better predictor of performance than the mean statistic (with a mean absolute percent error of 124%). The reason for this is that the mean statistic is skewed by outliers in the training set, whereas the median statistic is not. Figure 6-6A shows a histogram of percent-error using both mean and median statistics for the server www.asiasource.org. As can be seen, the mean statistic results in a skewed error distribution centered at 30%, whereas the median statistic results in one centered closer to 0%, as is expected.

Previous work [Wylie01] indicates that we need to be able to make predictions within a factor of 2 of the actual response times in order to be able to make good design decisions. Figure 6-7 shows a histogram of error across all servers. Figure 6-7A shows a histogram of the median absolute percent error for all servers, whereas Figure 6-7B shows the mean absolute percent error for all servers.

As can be seen, the median error is usually far lower than the mean error. This is because there are a few data points that are badly mispredicted (with a large error), but most of the points have a small error. In the vast majority of cases, we are well within a factor of two, and in fact, the median error is 26% on average.

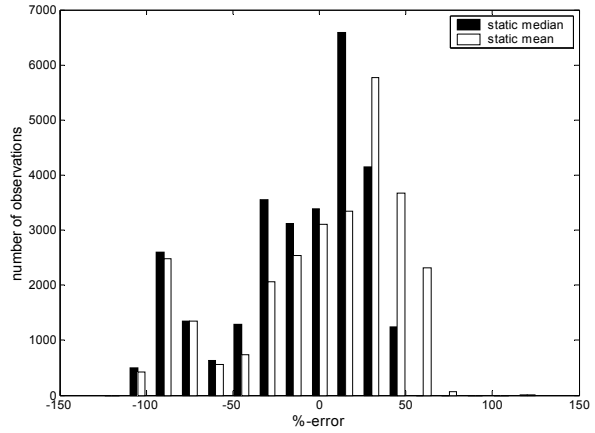


Figure 6-6A: Static model: median, mean

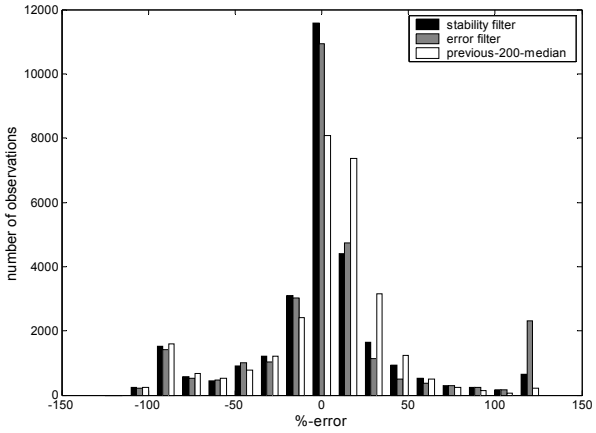


Figure 6-6B: Online model: stability filter, error filter, and previous-200 median

Figure 6-6: Histogram of mean error for server 19 (www.asiasource.org) using static (A) and online (B) models

Online model

As shown in the previous subsection, the static model is a reasonably good predictor of performance. In this subsection, we look at the online model and examine various algorithms to see which is the most effective in predicting performance.

An online model is useful because it allows us to take into account recent performance when attempting to predict response times. This is a great asset in many situations; for example, over a long (or even a medium-sized) length of time, servers' hardware, software, and network connections are periodically upgraded, causing changes in their performance characteristics. Making predictions based on recent history allow us to adjust to these changes. Also, during periods of heavy congestion, an agile predictive algorithm may allow us to get better measurements.

We evaluated four different predictive algorithms: (1) the median of the previous k measurements, (2) the mean of the previous k measurements. (3) The stability filter and (4) the error filter. For algorithms (1) and (2), we evaluated different values of k ranging from observations in the previous hour to observations in the previous month. Note that the results are significantly affected by having only one measurement every 10 minutes. More frequent sampling would result in lower error.

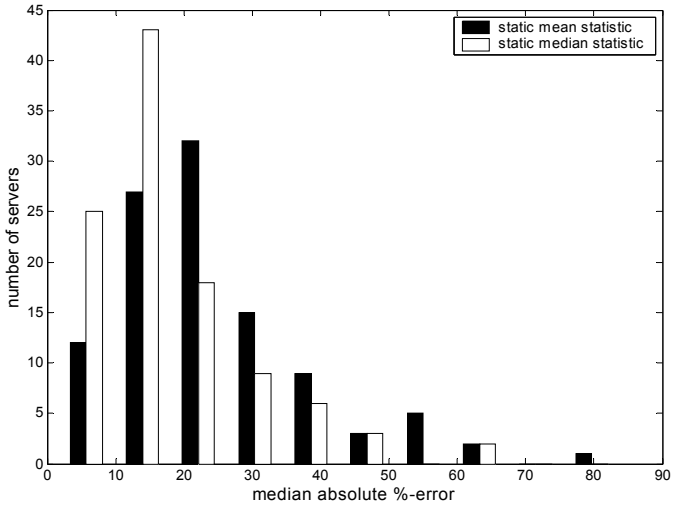


Figure 6-7A: Histogram of median absolute error

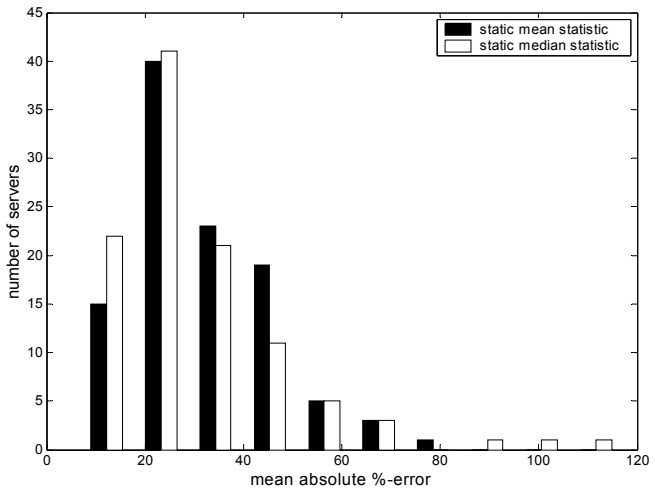


Figure 6-7B: Histogram of mean absolute error

Figure 6-7. Histogram of error for static prediction algorithms. Note that median absolute error is better than mean.

Values of k corresponding to a time period of less than one hour did not yield accurate predictions. This is because the predictions were too susceptible to noise. For all k>(one hour) we looked at, the median estimator was on average better than the mean. This is due to the same reason as in the static case: outliers tended to skew the mean unnecessarily.

It is generally true that smaller windows (i.e. values of k), result in higher standard deviations of the error. Since smaller windows require us to make predictions using less data, noise often causes our predictions to be very wrong, resulting in a higher standard deviation.

According to the data we have analyzed, for the previous k median predictive algorithm, the ideal value of k seems to be between 100 and 300, i.e. between ½ a day and 1-½ days. When using this metric, the previous k median function results in a mean absolute error of only 25%. The median absolute percent error of the previous-200-median function is 19%. The standard deviation of the error did not increase significantly over the static case.

Our experiments indicate that the stability filter is the best choice for an online model. Figure 6-6B shows the distribution of absolute percent error using online algorithms 1 (with k=200), 3 and 4, for the server www.asiasource.org. As can be seen, the stability filter provides the best results. These results are better than those obtained using the static model, pictured in Figure 6-6A.

Figure 6-8 shows the absolute percent error across all servers for three algorithms. Figure 6-8A shows a histogram of the median absolute percent error for all servers, whereas Figure 6-8B shows the mean absolute percent error for all servers. It is interesting to note that the error filter does well for the median absolute percent error metric, but does less well on average. This is because the error filter misses many data points by a very large margin, but is more often than not extremely accurate. The previous-200 median tends to miss all values similarly, and the stability filter is the best algorithm overall, with an average mean error of 26% and an average median error of 18%.

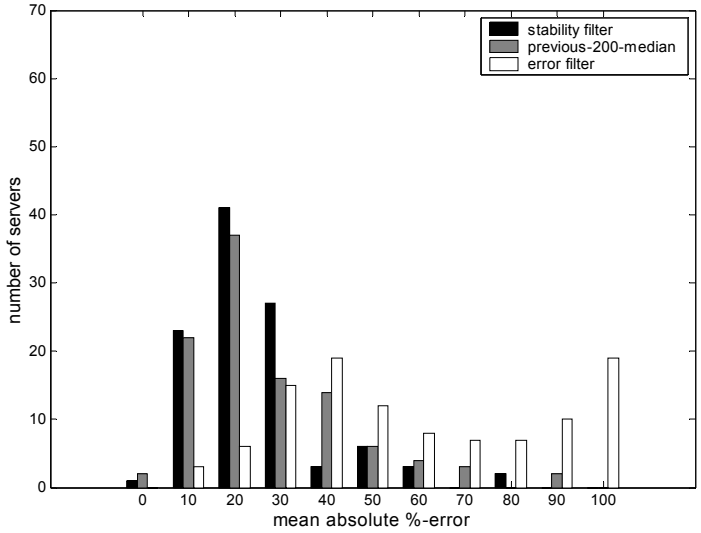


Figure 6-8A: Histogram of mean absolute error

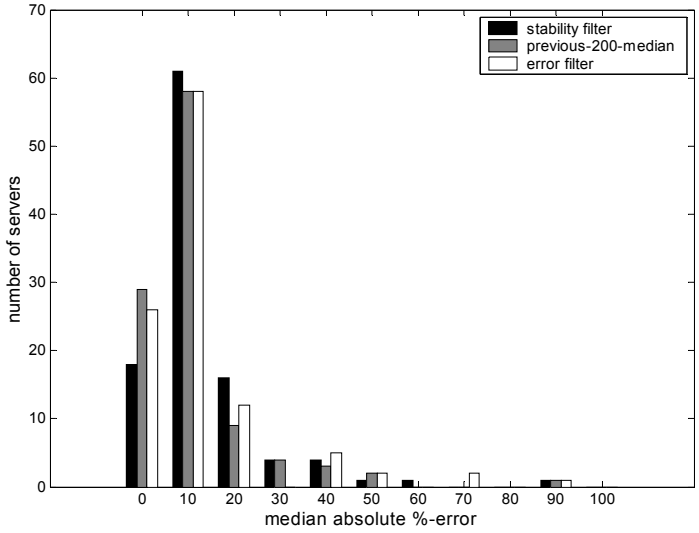


Figure 6-8B: Histogram of median absolute error

Figure 6-8. Histograms evaluating the performance of all online prediction algorithms across all servers. Note that most algorithms have significantly lower median error than mean error. This is because they often miss on a few outliers, which skews the mean error.

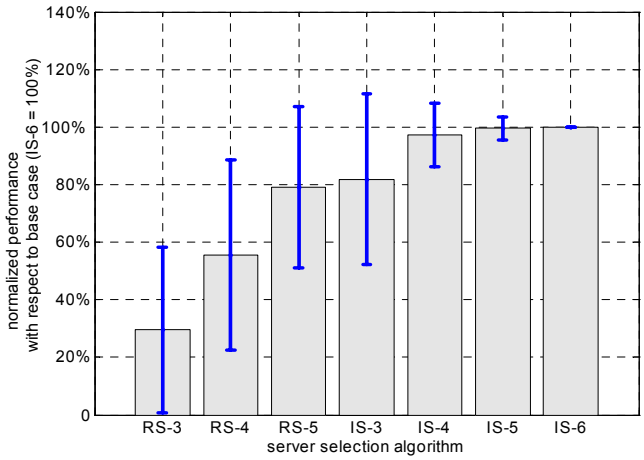


Figure 6-9A: This figure shows the average performance of each server selection algorithm normalized to the performance of the best case (r=6) over 50 server sets. The error bars denote one standard deviation of the relative performance. Note that on average, intelligent server selection algorithms are within 20% of the best case.

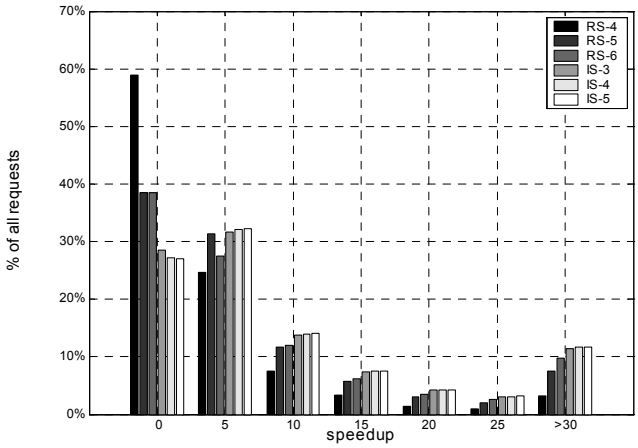


Figure 6-9B: This figure is a histogram that shows the average speedup (see Equation 7) of each server selection algorithm using RS-3 as the baseline. The x-axis is the value of the speedup and the y-axis is the percent of queries that achieved that speedup. Intelligent server algorithms on average show speedups of over 10.

6.4 Experimentation and Results

6.4.1 Single client analysis using a model

Using our model and the trace data we collected, we evaluated the over-requesting and intelligent server selection strategies described earlier.

Effectiveness of various over-requesting algorithms

In order to show the effectiveness of various over-requesting algorithms, we selected 50 sets of six traces, and used our online WAN model to predict behavior for all possible intelligent and random server selections for a n=6, m=3 scheme. The traces were used to supply the clients with server response times.

Figure 6-9A shows the average performance for each algorithm normalized to the best possible request scheme, which is "request all n shares" (in this case, r=6). "IS" represents intelligent server selection, and "RS" represents random server selection. The value after the algorithm name is the value of r. On average, RS-3's performance is only 30% of the best case. However, intelligent server selection algorithms on average achieve performance that is within 20% of the best case.

Figure 6-9B is a histogram that shows the speedup that each server selection algorithm achieves over RS-3. High speedup values are desirable. Intelligent server selection algorithms with over-requesting offer higher speedups (over 10x the baseline, on average) than random over-requesting with similar values of r (2x, on average). Speedup for algorithm *algo* is defined as:

$$speedup_{\text{algo}} \equiv \frac{throughput_{\text{algo}}}{throughput_{RS,r=m}} \qquad (6.7)$$

Model accuracy

We also examined the error of the model's predictions. The model is able to predict the throughput of over-requesting schemes for various sets of servers quite well, generally with mean absolute percent errors under 30%, and median absolute percent errors under 25%. The mean error is far higher than the median error because of the effect of outliers. In many cases, there is a very small fraction of predictions that

	rs-3	rs-4	rs-5	rs-6	is-3	is-4	is-5	is-6
mean abs error	48%	31%	30%	30%	36%	34%	32%	30%
median abs error	33%	21%	21%	21%	17%	20%	20%	21%
abs error std	69%	36%	35%	32%	86%	56%	33%	32%

Table 6-3: This shows the average prediction error of the online model, for various selection algorithms. These values are the averages of values obtained from 50 different 6-server sets.

miss the actual throughput by a few orders of magnitude. Although most of these prediction errors are unavoidable because they are the result of variation in individual server response times, they will not have a serious impact on an actual system's performance due to their rarity.

Table 6-3 shows the prediction errors for various over-requesting schemes for an n=6, m=3 scheme averaged over 50 different sets of servers. Note that both the prediction error as well as the error standard deviation decrease as the number of shares requested increases. This confirms our hypothesis that over-requesting will help mitigate performance penalties primarily through exploiting variability in the servers.

6.4.2 Server selection strategies with multiple clients and network delay

We use the simulator described in section 6.2.3 to examine the global effects of multiple clients employing server selection strategies. In sections 6.4.2 and 6.4.3, we discuss two example simulation systems, both of which have n=6 and m=3. The six servers' processing times are 8, 10, 12, 15, 19, and 24 ms respectively. Each client's think time is an exponential random variable with a mean of 100 ms. Data is collected over 100 simulated seconds. The server's maximum queue depth is 80, and there is a network delay of 52 ms, exponentially distributed. In system A, the server processing times are constant, and in system B, they are exponentially distributed. The clients use the stability filter as the predictive algorithm.

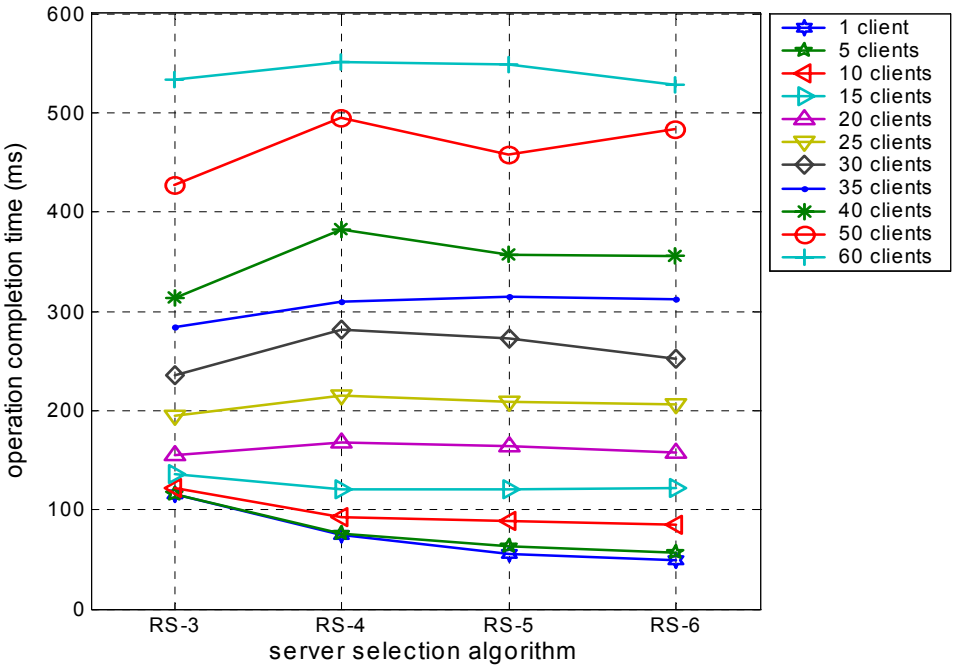


Figure 6-10: Random over-requesting with multiple clients, system B. Lower numbers on the y-axis are better.

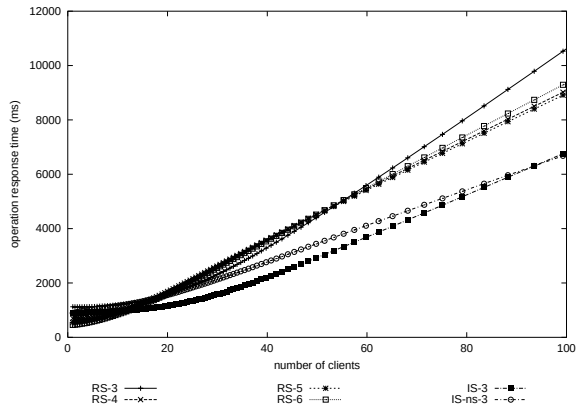


Figure 6-11A. System A

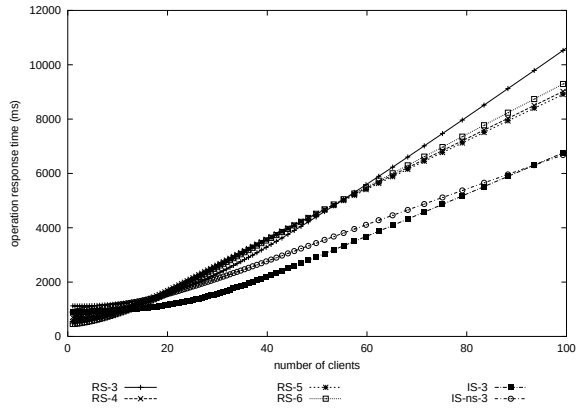


Figure 6-11B. System B

Figure 6-11. Comparison of Intelligent server selection with and without random sampling. The y-axis is mean response time for the operation as seen by clients (lower numbers better). IS-ns represents Intelligent server selection without sampling.

Random over-requesting

We first examine the effects of many clients employing *random over-requesting* to retrieve data stored in system B. We ran this experiment with 1, 5, 10, 15, 20, 25, 30, 35, 40, 50 and 60 clients in the system. The results are shown in Figure 6-10. The x-axis shows the value of r (remember that $n \geq r \geq m$, so $6 \geq r \geq 3$), and the y-axis shows the average transaction completion time.

When we have a small number of clients present, random over-requesting has the intended effect of decreasing the transaction time. Since the servers are mainly unloaded, the extra requests that are made do not cause queuing delays. Since the servers' processing times are the result of a random variable, waiting for the third slowest server instead of the slowest server allows for improved performance. Note that as the number of clients in the system increases, the operation times increase. These increased times are due to queuing delays caused by higher load on the servers.

When 15 clients are present in the system, RS-5 and RS-6 offer no increase in performance over RS-4. This effect is due to the additional requests that are sent out; clients using r=5 and r=6 are sending so many extra requests that the queuing delays are masking the performance gains from waiting for the 3rd and 4th slowest servers instead of the 2nd. Average queue lengths that were recorded during this experiment support this conclusion. For systems with more than 15 clients, r=3 is the best choice, again due to queuing delays. When 60 clients are present in the system, the operations with lower values of r experience many time-outs. Since algorithms with higher r are less likely to suffer a timeout, they begin to perform better against RS-3. In fact when 200 clients are in the system, RS-3 once again performs worse than RS-4 and RS-5.

To summarize what we have seen by analyzing random over-requesting, it is evident that in a situation where a client makes decisions purely to improve its own performance without regard for the global implications of its decision, it will always request from all six servers. In a case where it is the only client doing this, it will reap improved performance. This is an illustration of the tragedy of the commons; from each client's perspective, requesting from all six servers has a very low, almost nonexistent negative repercussion, but the possibility of a very valuable decrease in the transaction time. Just as in the commons in old England, if all clients begin acting in a similarly greedy manner, the servers become much slower, and the 'commons' become spoiled for everyone.

Intelligent Server Selection

As described in Section 6.2.3, the clients in the simulator can independently implement intelligent server selection. To demonstrate the need for the periodic sampling of 'unintelligent' servers, we conducted an experiment using intelligent server selection both with and without the sampling. We examine the algorithms' performance in both systems A and B. A plot of the average response time against similar values obtained from random over-requesting is shown in Figure 6-11. Intelligent server selection without sampling is labeled "IS-ns" and intelligent server selection with sampling is labeled "IS". We can see for systems with fewer than 40 clients, intelligent server selection without sampling is less effective than intelligent server selection with sampling. This occurs because clients have only stale data about servers, and will not switch from their current servers unless the current set of servers becomes slower than the stale response time predictions for the other servers. Of course, since the information is stale, the client has a more pessimistic view of the other servers than necessary. In the same graph, it can be seen that intelligent server selection with sampling generally performs better than either random over-requesting or intelligent server selection without sampling. Figure 6-12 shows the total response time (including queuing time and processing time) that each server exhibits throughout the experiment. After the system stabilizes, the fastest servers see the highest request rate, and the slower servers see progressively fewer requests, resulting in all servers exhibiting a similar response time. This is exactly the situation that we want; each server is processing a number of requests in line with its capacity to service those requests. Here, it can be seen that the system eventually settles to a state where all the servers exhibit approximately the same response time when possible.

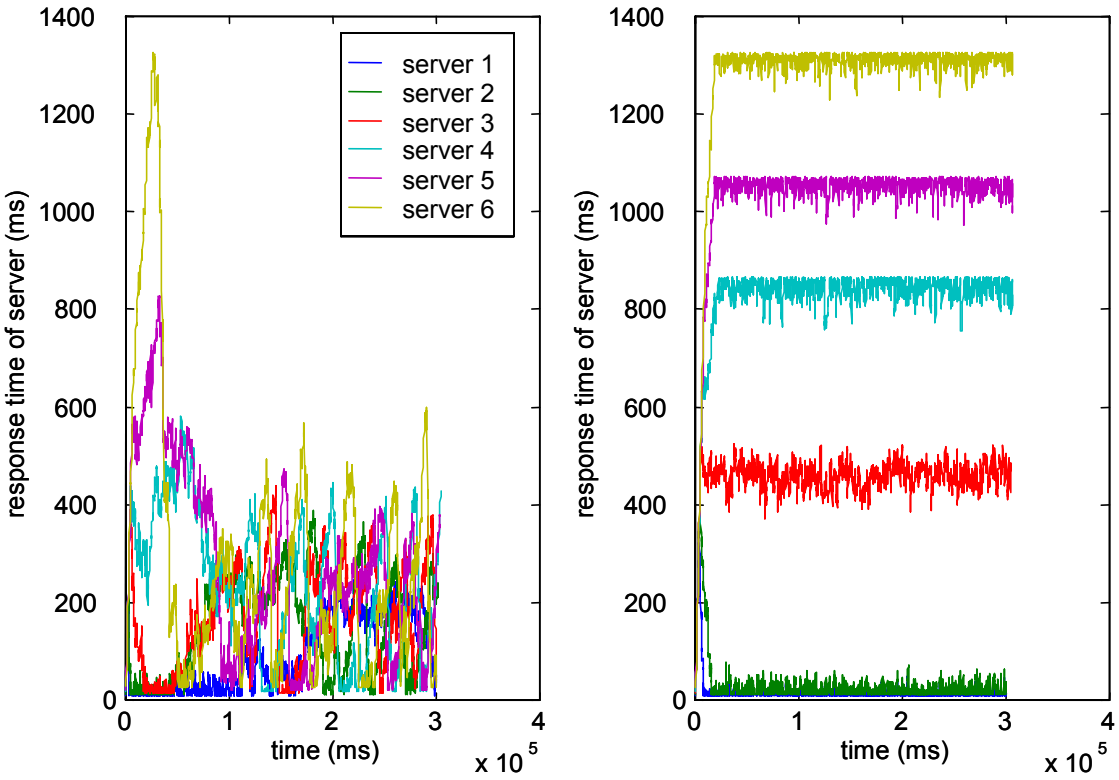


Figure 6-12. This is a plot of server response time versus simulation time, using algorithm IS-3 (on the left) and RS-5 (on the right). Servers 1 through 6 have processing times that are 8, 10, 12, 15, 19, and 24 ms respectively. Note that with RS, Each server's response time is fairly consistent and staggered, whereas with IS, response times of all servers are similar. This shows that IS allows servers to handle load according to their ability, leading to a more optimal global situation. These data were generated through simulating a 40-client system.

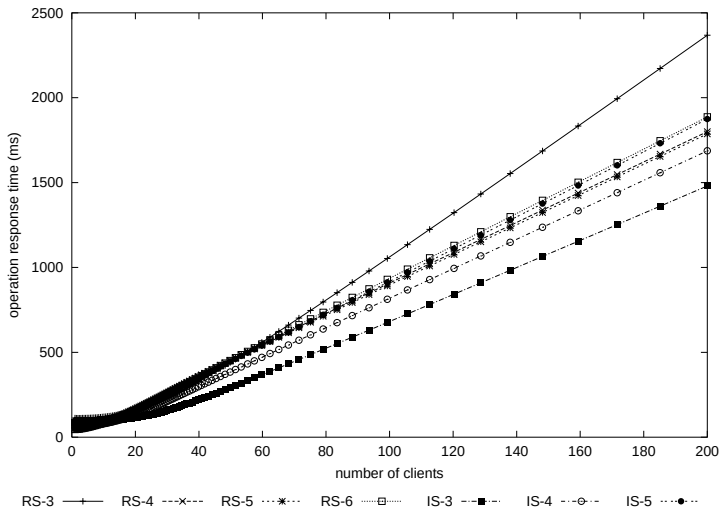


Figure 6-13A: Performance of algorithms over large range of clients, System A.

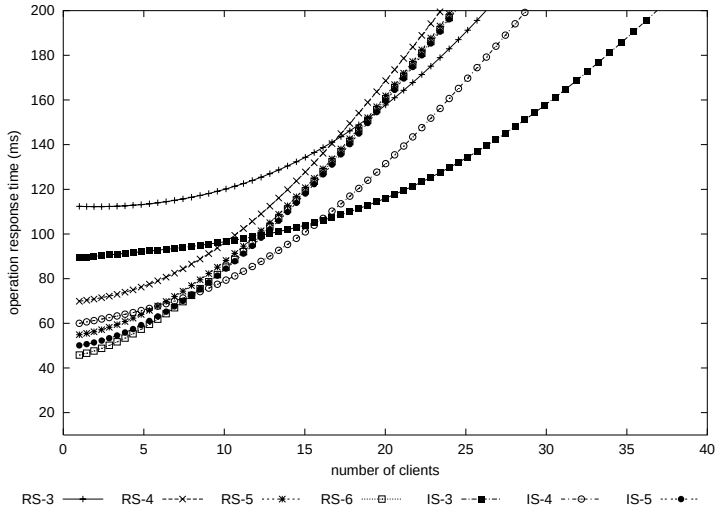


Figure 6-13B: Performance of algorithms over small range of clients, system A.

This trend is especially stark when compared with the behavior of random server selection (also in Figure 6-12). There, each server receives the same number of requests. This results in the slower servers being overloaded and the faster servers sitting mainly idle. That situation is not desirable. Figure 6-11 also demonstrates the IS algorithm is quite scalable with respect to the random schemes. The one drawback to the IS algorithm is the fact that each client will on average make an extra sampling request to each of the slower servers once every 6 requests. When there are many clients, this results in the slowest servers timing out even on some sampling requests. If the frequency of extra requests is reduced, this effect is mitigated, but clients are forced to operate with stale data. Evaluating different sampling frequencies is left as future work. By using intelligent server selection with sampling, clients independently making server selection decisions allow the system to settle to a globally good state even without any centralized control.

Comparing intelligent over-requesting and random over-requesting

The same systems A and B are now used to compare performance of intelligent server selection and random server selection across all values of *r*. Figures 6-13A, 6-13B, 6-13C and 6-13D show results for up to 200 clients. Note that when there are few clients in the system, algorithms that use a high *r* perform very well. This shows that over-requesting is able to capitalize on the variation in the network delay to improve performance.

As the number of clients increase to 40, however, intelligent server selection with larger values of *r* slowly become less valuable because of the extra load they place on the servers. RS-5 and RS-6 start out doing quite well, but with more clients in the system, they too begin to perform poorly. In fact, they are outperformed by all the intelligent server selection algorithms. RS-3 starts out poorly, but scales well, mainly because it places no excess load on the servers. IS-3 is by far the best algorithm for a system with 40 clients. Note also that intelligent server selection outperforms random selection schemes where both have the same value of *r*.

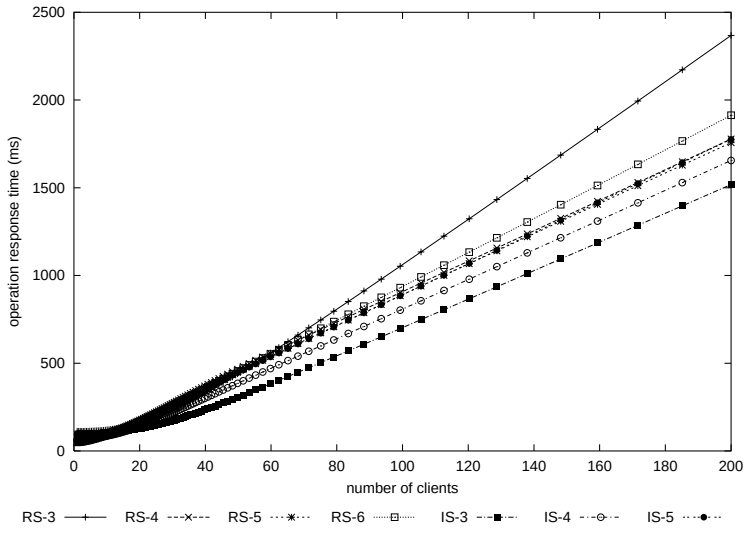


Figure 6-13C. Performance of algorithms over large range of clients, System B.

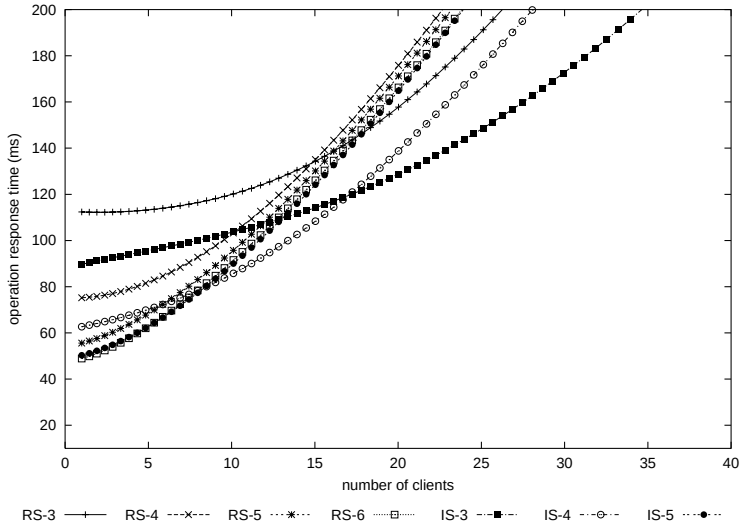


Figure 6-13D. Performance of algorithms over smaller range of clients, System B.

As the number of clients increases past 60, algorithms with lower values of *r* such as RS-3 again start performing significantly worse than algorithms. This is because RS-3 is extremely susceptible to timeouts, which significantly increase the average transaction time. With 60 clients in the system, over-requesting starts to become increasingly attractive because its ability to mask failures outweighs the added load placed on the servers. The IS-3 algorithm continues to outperform all others.

Algorithms in system B perform similarly to those in system A since the variance in the servers' processing time is not very large when compared to that of the network delay. Interestingly, however, with many clients, intelligent server selection performs better than in system A. Due to the variation in server processing times, it becomes more likely that some of the 'unintelligent' servers that IS periodically samples will respond more quickly than the intelligent ones, giving an unexpected performance benefit. It is interesting to note that the variance in the server processing time allows algorithms with higher *r* values to offer better performance than in system A, and thus causes those with lower *r* values (including intelligent server selection) to do comparatively worse.

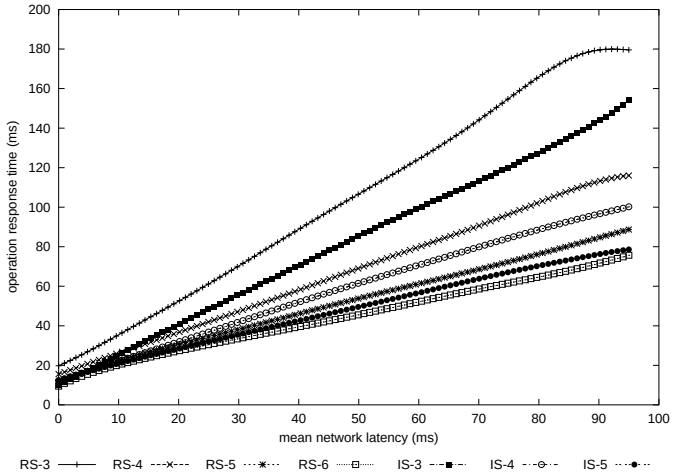


Figure 6-14A. This system is based on system A, but with different network latencies. The network latency is exponentially-distributed Note that intelligent server selection with *r*=C often offers comparable performance to random server selection with *r*=C+1. This is an important performance improvement.

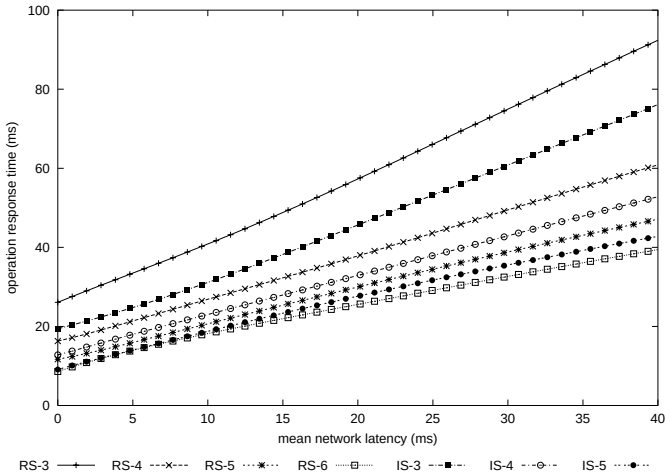


Figure 6-14C. Performance of various over-requesting algorithms with respect to increasing network latency. This is based on system B.

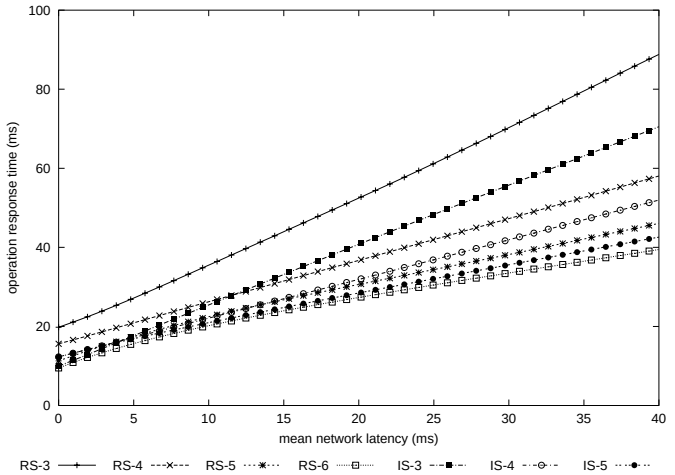


Figure 6-14B. Same as above, for fewer clients. Note here that when x=0, all intelligent selection algorithms perform equally well, with a best-possible transaction time of 12 ms, the 3rd fastest server.

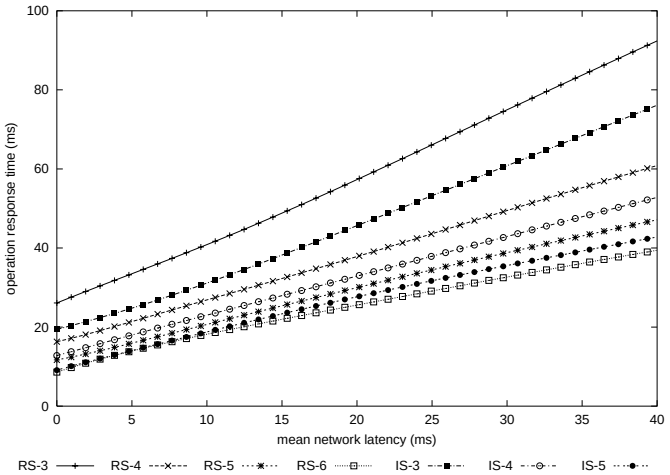


Figure 6-14D. Same as above, but for fewer clients. Because the server processing time is variable, at x=0 intelligent server selection algorithms with higher *r* perform better.

6.4.3 Server selection strategies with a single client

In this section, we look at the effect of network latency on the performance of different server selection strategies used by one client. The first situation we examine is varying the mean network latency from 0 to 90 ms, both on system A and B. In all cases, the network latencies are exponentially distributed. Figure 6-14A and 6-14B shows the results. In system A, for low mean network latency values, the intelligent server selection algorithms outperform random server selection schemes. When the network latency approaches 10 ms, IS-3 no longer outperforms the RS-4. Similarly, IS-4 no longer outperforms RS-5 after 15 ms. This occurs because the network latency becomes a larger component of the response time than the server processing time. Since the network latency is random, it becomes more difficult for the clients

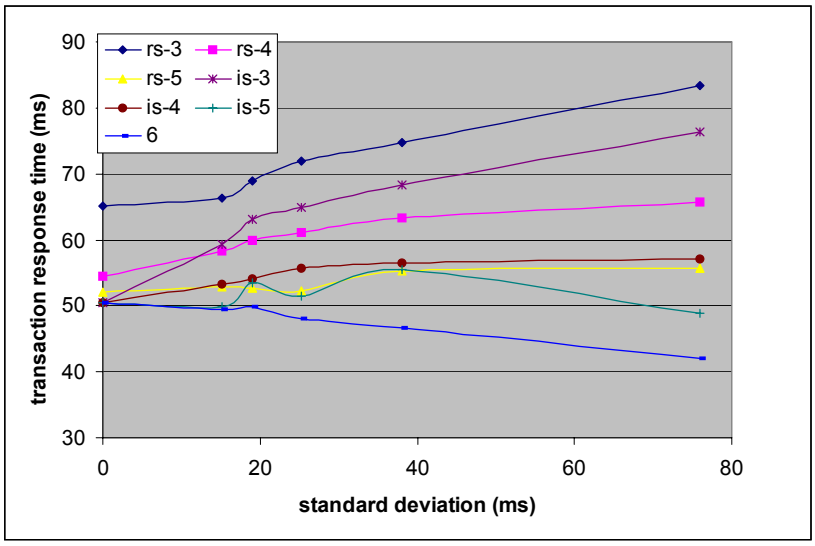


Figure 6-15A. Performance of various algorithms with respect to network delay's standard deviation. The mean network delay for this graph was 38 ms. Based on System A.

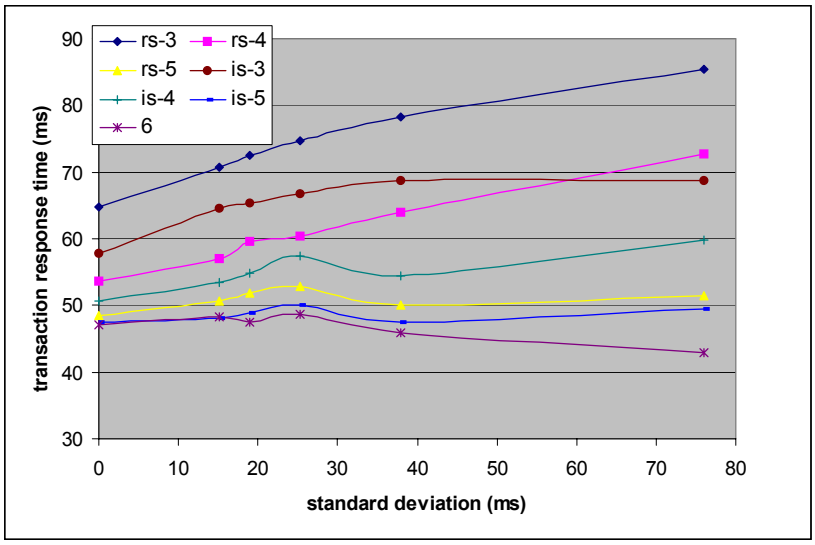


Figure 6-15B. Performance of various algorithms with respect to network delay's standard deviation. Similar to above, except based on System B.

to predict future response times and thus intelligent server selection becomes less valuable. When similar latencies are added to system B, the intelligent server selection algorithms degrade more quickly than in system A. This is because the added randomness from the server processing time makes the servers response times even less predictable.

Next, we vary the standard deviation of the network latency between 0 and 80 ms while keeping its mean fixed at 38 ms. Figures 6-15A and 6-15B illustrate the results. In system A, the performance of RS-*r* is slightly worse than IS-*r* for large variances. When the network latency is stable (standard deviation = 0), all intelligent algorithms do equally well. In addition, as the standard deviation increases, the transaction time using RS-6 approaches the best possible time of 40ms.

In system B, we see that the IS-*r* algorithms do not scale better than RS-*r* algorithms, once again because the exponential server processing times makes the response times more difficult to learn. However, RS-(*r*+1) still significantly outperforms IS-r. Again, as the standard deviation increases, the transaction time using RS-6 approaches the best possible time of 40ms.

The final scenario we examine shows the ability of over-requesting to reduce variance. We examine a system in which n=24 and m=6. Servers' processing times range from 8ms to 50 ms, and the network latency is normally-distributed with a mean of 40 ms and a standard deviation of 40 ms. The results of intelligent server selection and random server selection are shown in Figure 6-16. The error bars on each line are one standard deviation in the operation times as observed by the client. Intelligent server selection provides better response times, and lower variances. Over-requesting lowers both the mean operation time as well as the variance of the observed operation completion times.

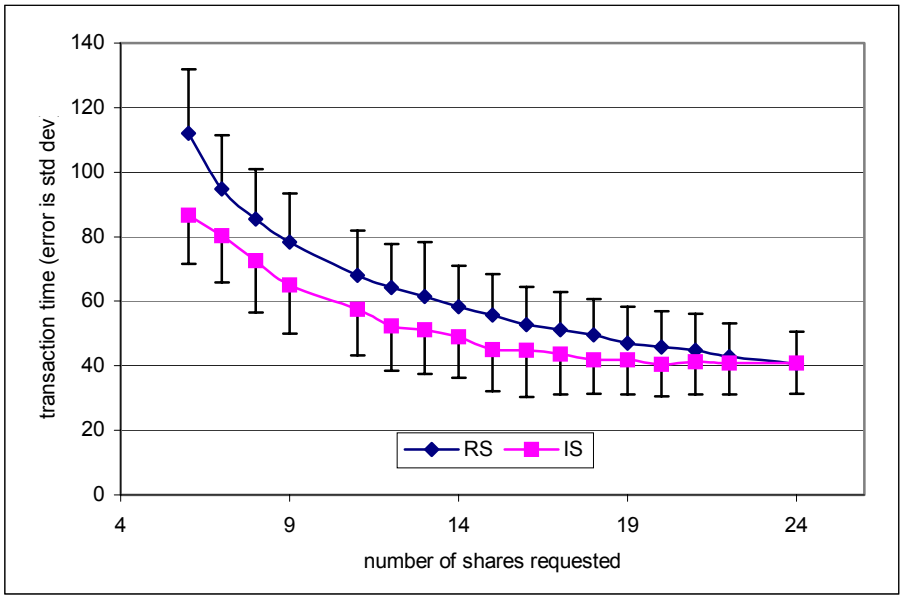


Figure 6-16. One client random over-requesting in a scheme with n=24, m=6. *r* is on the x-axis. Servers' processing time distributed between 8 and 50ms. The network delay was normally-distributed with a mean and standard deviation of 40ms. The error bars show one standard deviation in the client's recorded response time. The bars are only in one direction to prevent them from overlapping and making the graph too difficult to read.

6.5 Conclusions

6.5.1 Predictability of individual web servers in a wide area

As part of this work, we collected response times at 10-minute intervals for 106 servers over an 8-month period. We evaluated the predictability of these values by using both a static model as well as an online model. With a static model, where we generated a summarized statistic using 20% of the data, we used one unchanging model of the server and compared it against the remaining 80% of the data. Our algorithm was able to predict performance with an average error of only 32%, and a 50th percentile error of 26%. Using an online model, we evaluated four different algorithms, the last-200-median, which involved taking the median of the previous 200 measurements, the last-200-mean, the stability filter [Kim01], and the error filter [Kim01]. We show that the online model allows us to predict performance with an absolute mean percent error of less than 25% using the stability filter. This shows that individual server performance is indeed predictable.

6.5.2 Lack of correlation between pairs of web servers

We also analyzed correlation between the performances of all pairs of servers. Over this 8-month period, only four of the roughly 5000 pairs of servers showed medium-to-high correlation. These servers shared nearly the entire end-to-end route.

6.5.3 Random over-requesting vs. intelligent server selection

We use the large set of collected data as traces, and show that we can predict performance of and properly choose between different levels of over-requesting and intelligent server selection. As shown by Wylie et al. [Wylie01], we are well within the factor of two that allows us to make proper scheme selection decisions. We show that random over-requesting almost never outperforms intelligent over-requesting when both selection schemes use the same *r*. When only a few clients are in a system, over-requesting signifi-

cantly improves both the performance (response time) as well as the stability of the performance. Over-requesting helps to mitigate variability in the response times by insulating us against temporarily slow servers. When too many clients begin to adopt over-requesting, the tragedy of the commons problem [Hardin68, Turner93] surfaces and the servers' performance degrades.

6.5.4 Which server selection algorithm to use; which *r* value is best?

The most important conclusion that should be taken from this paper is that it is very difficult for a client to determine the level of *r* it should use with the limited information it can directly observe. It is generally true that random over-requesting is almost never a good policy. In nearly every case we examined, intelligent server selection was never significantly worse than random over-requesting when they both had the same value of *r*. Since it has been shown that in most situations, servers' response times are reasonably predictable, this makes sense.

If the servers are mostly idle, intelligent server selection with over-requesting is extremely useful; it allows a client to add insurance against failures, slower servers, and variance across servers. However, if the servers cannot handle the additional load that this strategy generates, requesting more than *m* shares could unnecessarily slow the operations down. Thus, when the servers are not idle very often, the optimal strategy is IS-*m*. The strategy the client should use in a given situation will depend heavily on how able the server is to handle the expected additional load from using that server.

Unfortunately, the client will not be able to see any change in the servers' performance based on its own over-requesting decisions since servers are only affected by the collective actions of many clients. Thus, it is difficult for the client to determine under what load the server is operating and its ability to handle additional requests. Clients may therefore not be able to gather enough information to make sound decisions about what value of *r* to use on their own. This makes IS-*r*, where *r* is very close to *m*, the safest policy to use.

6.5.5 Future work

There are a few possible solutions that could allow clients to determine what level of *r* to use. One is for server administrators to periodically distribute to clients historic server load data, the average number of clients serviced, and the additional number of requests it could handle. Since that policy would require significant human intervention, files containing this content could be automatically updated and stored on the servers, so clients could periodically retrieve it. With this information, clients will be able to individually determine the value of *r* that may be used without overloading the system.

Another solution involves clients placing various priorities on their requests such that the *m* fastest servers receive requests with the highest priority, while the other *r*-*m* servers would receive requests of lower priority. The servers would process requests only when there are no requests of higher priority waiting. This way, over-requesting would not be able to significantly hurt different servers' performance. Unfortunately, this would involve making modifications to server software, which we wished to avoid.

Future work will concentrate on simulating these two solutions and determining the benefits and drawbacks of each.

The simulator discussed in Section 6.2.3 can be expanded to more accurately model a distributed system in a WAN. The simulator will need to allow certain servers and clients to behave as if they are "close" to one another and others farther away, to approximate a network topology. The simulator would imitate this situation by having different network delays for each server/client pair. Additionally, exploring situations where server information is shared between clients could yield interesting results. The servers should also drop requests with a certain failure probability even when the queue is not full in order to simulate unavailability. Bakkaloglu et al.'s work [Bakkaloglu02] can be used in the simulator by causing server failures to conform to values from a correlation parameter. Finally, the simulator should also be able to

model varied size requests, and possibly model various server-side queuing algorithms including shortest job first.

Finally, implementing the server selection algorithms described in this paper in a prototype survivable storage system such as PASIS would provide valuable insight and allow us to evaluate our simulations of multi-client systems.

7 TRADE-OFF ANALYSIS IN PASIS

The previous three sections described our modeling effort to capture three essential system properties for survivable storage systems—performance, availability and security. These models constitute a foundation based on which we can begin to model our trade-off analysis. Being able to make informed trade-offs among system properties that are not necessarily commensurate with each other is important for system-level design decisions. Measurements from our initial prototype were used to configure the default parameters and to validate the performance components of the model. The approach and its use have been published as "Selecting the Right Data Distribution Scheme for a Survivable Storage System" [Wylie01], a CMU/SCS technical report available on the PASIS web site (http://www.pdl.cmu.edu/PASIS/).

To illustrate the trade off analysis, again we consider the three-dimensional trade-off space with performance, security, and availability as its three axes. Figure 7-1 shows such a space and depicts seven data distribution algorithms with respect to the three axes in the space.

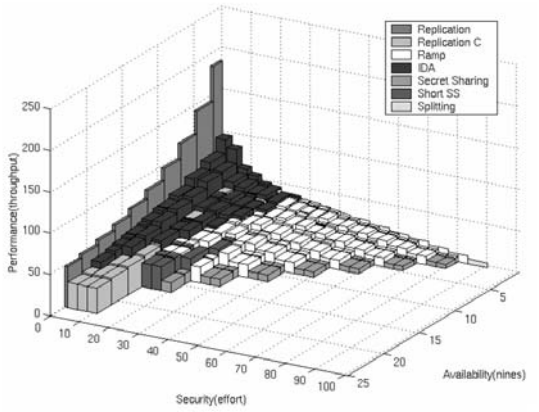


Figure 7-1. Data distribution scheme selection surface plotted in trade-off space.

The data points of the various data distribution algorithms are measured in a system with 10 storage nodes. Performance is quantified as the number of 32 KB requests per second that can be satisfied for a single client. Only the best performing scheme for a given security and availability level is shown. The default value for the network parameter is 100 Mbps. Security, again, is the effort required to compromise the confidentiality or the integrity of stored data. Availability is the probability that stored data can be accessed, and is measured in "number of nines." The default value for the system workload is a read/write probability of 0.5 (an equal number of reads and writes). The default value for a single storage node's failure probability is $f_{node} = 0.005$. We will refer to this system configuration as the default configuration.

To compare scheme selection surfaces, we use a metric with two components. The first component quantifies what percentage of the scheme choices change. The second component quantifies the magnitude of the performance cost associated with using the other surface's scheme at a given point. For example, we state such results as "the selection surface differs over X% of the area with an average difference of Y%." When performing sensitivity analysis, we reverse the direction of the comparison (i.e., we examine the impact of inaccuracies in the assumed system configuration). This allows us to determine how accurate the configuration representation must be for the trade-off analysis to be useful.

In some regions of the trade-off space, many schemes have similar performance over a range of configurations. In such regions, different schemes may be selected for different configurations, but with minimal performance impact. In other regions, significant performance costs are incurred if the wrong scheme is selected. To capture this interesting effect, we sometimes perform a comparison between two selection

surfaces and only list the area and difference for regions that have changed by some minimum amount (i.e., we ignore differences of less than 10% to get a better measure of large scale effects).

Another method of analyzing the trade-off space is used to understand the consumption of resources by the system. Specifically, we examine the ratio of time spent performing CPU operations to the time spent performing network I/O in order to ascertain what regions of the selection surface are bound by each resource. A scheme is considered to be bound by a resource if the resource accounts for greater than 80% of the overall performance time. The resource consumption for the default configuration is presented in Figure 7-2.

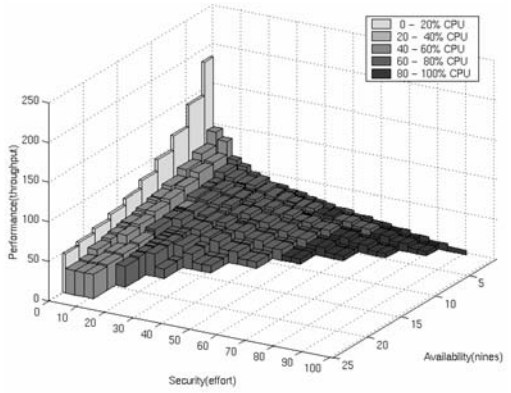


Figure 7-2. Balance of resources in default configuration: Light gray indicates a region of the surface in which the selected scheme is bound by the available network bandwidth. Dark gray indicates a region in which the CPU binds the selected scheme. In the default configuration, replication (which appears along the zero security line) is network bound. Farther down the security axis, schemes are CPU bound. There is a large region, comprised of information dispersal, replication with cryptography, ramp schemes, and short secret sharing, with balanced resource consumption.

7.1 The Model Sensitivity

If the selection surface is highly sensitive to small changes in the configuration, then it might not be a useful design tool because it would require exact information about how a system will behave. To ensure that the selection surface is relatively stable, we examined its sensitivity to the system characteristics and workload aspects of the default configuration. For the system characteristics, we varied the default network speed by a factor of two in each direction. For the workload, we considered workloads of 40% reads as well as 60% reads. Considering these cases, the selection surface differs by less than 9% of the area with a difference of less than 9%. From this, we conclude that the selection surface is relatively stable. As well, this means that as long as system characteristics are modeled with moderate accuracy and the workload is roughly understood, the scheme selection surface can provide significant insight into the real configuration.

System Characteristics

To investigate the impact that system characteristics have on proper scheme selection, we fix the CPU speed and vary the network speed. This explores system configurations representative of a wide range of distributed systems such as peer-based computing on a LAN or client-server architectures.

We begin by varying the default network speed one order of magnitude (i.e., to 10 Mbps and 1 Gbps). Speeding up the network has little effect on the selection surface. The selection surface differs over less than 10% of the area with an average difference of 3%. Changes resulted from replication with cryptogra-

phy, a computationally cheap and space-inefficient algorithm, becoming a better selection than information dispersal. Figure 7-3 shows the selection surface for the fast network. When the network bandwidth is reduced, the selection surface differs over 26% of the area with an average difference of 17%. Clearly, this change is substantial. However, some of this region is comprised of a "checkerboard," where the right choice bounces between two schemes with little performance difference. In this checkerboard region, the selection surface differs over 11% of the area with an average difference of 3%. In one part of the checkerboard region, short secret sharing and ramp schemes have similar performance. In another part, information dispersal schemes with parameters that make them more space-efficient are selected over less space-efficient ones. The remainder of the difference between the surfaces is due to 14% of the area with an average difference of 29%. In this region, which is along the availability axis, information dispersal dominates replication. Figure 7-4 shows the selection surface for the slow network.

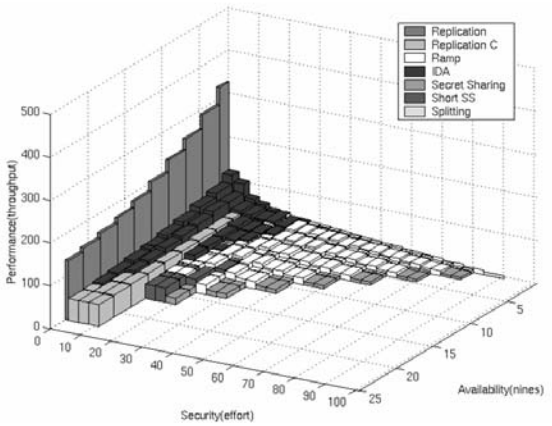


Figure 7-3. Fast network (1 Gbps): Replication with cryptography dominates the low security, high availability region of the graph, because the network consumption of replication has a low performance cost on a fast network.

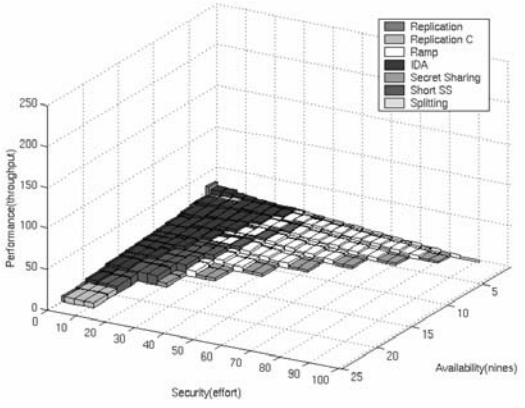


Figure 7-4. Slow network (10Mbps): Information dispersal dominates along the availability axis. Short secret sharing has similar performance to ramp schemes in the foreground of the graph.

An analysis of the resource consumption over the selection surfaces for the various network speeds considered shows that, for the fast network most of the selection surface is CPU bound, for the default model most of the surface is balanced, and for slow networks most of the surface is network bound. Many con-

clusions can be drawn from the results of these experiments. First, scheme selection depends on the balance between the speed of the processor and the speed of the network. Second, in system configurations that include a slow network, the space-efficiency of a scheme is a significant indicator of its overall performance. Third, the default configuration can be used to select schemes for faster networks since scheme performance is generally CPU bound in the default configuration.

Impact of Workload

To investigate the impact that workload has on scheme selection, the ratio of reads to writes was varied from a 100% read workload to a 100% write workload. Recall that the default workload is equal parts reads and writes. As shown in Table 7-1, the experiment found that scheme selection is insensitive to workload over a large portion of this range, 10% reads to 90% reads. However, at the end points of the range, there is substantial change in the selection surface. In particular, changing the workload can change the operating region of an algorithm (i.e., it changes the availability guarantee that can be made)—a clear difference from modifying system characteristics.

Read Workload (% reads)	Surface Change	Average Performance Cost
0%	97%	33%
5%	67%	6%
20%	22%	5%
10%	6%	4%
60%	4%	2%
80%	26%	7%
95%	50%	16%
100%	61%	20%

Table 7-1. Impact of workload on selection surface: Scheme selection is insensitive to changes in the mid-range of workloads. Significant changes occur near the endpoints of the possible workloads.

The reason that the selection surface is insensitive to the workload is because the workload mainly affects system performance. As the read workload increases/decreases, the selection surface expands/contracts along the performance axis. The reason for this is that the read (decode) costs are lower than the write (encode) costs for most algorithms. Only at the ends of the workload range do the different rates at which schemes are able to scale result in new schemes outperforming the scheme selected for a mid-range workload. The conclusion of this experiment is that, for most read-write workloads, scheme selection can be done assuming a workload of 50% reads because the selection surface is stable over a large range. However, for write-once/read-many storage systems the trade-off space differs radically.

The selection surfaces for the 100% read and 100% write workloads have some specific features that provide insight into the trade-off space. Figure 7-5 and Figure 7-6 show the selection surfaces for these workloads. Splitting dominates the low availability-high security region (the back plane of the graph) for the 100% read workload, and it occupies the diagonal that demarcates the edge of the operational region for the 100% write workload. For other workloads, splitting only occurs at the maximum-security point. Splitting's decode operation is faster than secret sharing's; however, for any other workload secret sharing dominates splitting because splitting has extremely poor read availability. Splitting's decode operation is extremely fast compared to its encode. This is why it is selected for the read workload—its encode performance does not penalize it.

This insight about splitting is applicable to the rest of the threshold schemes. For the 100% read workload, schemes are not penalized for poor write performance. The result of this is that ramp schemes dominate the region held by information dispersal for write dominated workloads—ramp schemes with large

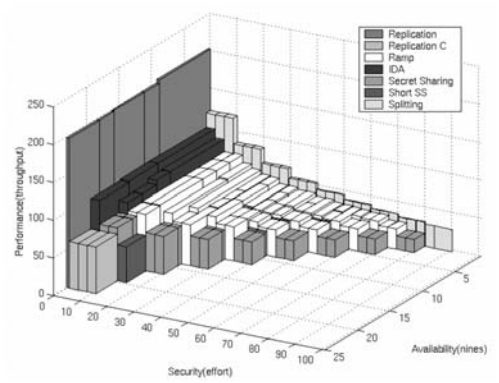


Figure 7-5. 100% Read workload.

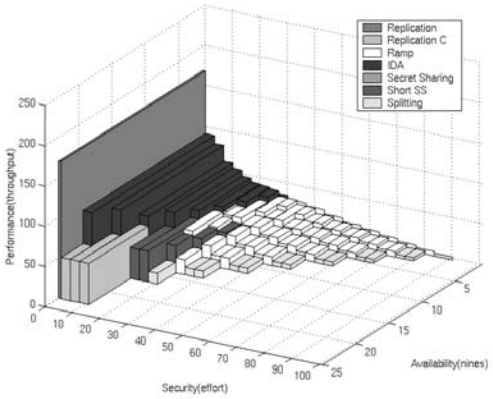


Figure 7-6. 100% Write workload.

n and low *m* are competitive with information dispersal schemes with mid-range *n* and similar *m* values because their poor write performance is counted. Also, short secret sharing is shown to have little value for a read-only workload. Short secret sharing offers good security for a low encode cost. Again, the encode cost does not matter in a read-only workload—ramp schemes offer better read-only performance than short secret sharing.

Considering the resource consumption of the 100% read and 100% write workload is also instructive. Figures 7-7 and Figure 7-8 demonstrate that the best performing schemes have a balanced utilization of resources when performing reads, whereas the CPU is the scarcest resource when encoding data for high security. The fact that security is CPU bound matches intuition—security is expensive and system designers loathe paying the price. However, the fact that reading secure data is not as expensive, in terms of CPU cycles, is significant. Indeed, for workloads with >60% reads the utilization of resources in the system is balanced deep into the security region.

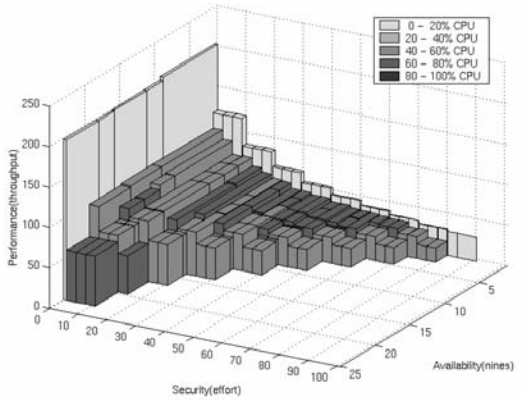


Figure 7-7. Balance of resources for a 100% read workload

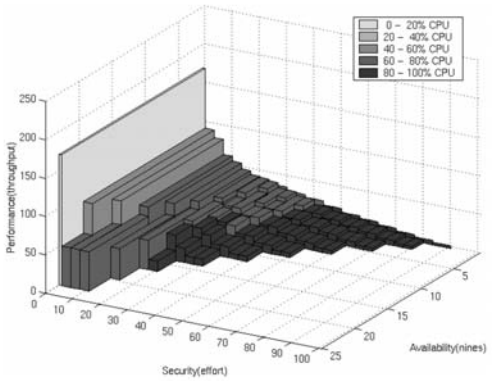


Figure 7-8 Balance of resources for a 100% write workload

7.2 Analysis

Although the trade-off space is complex, we expected specific algorithms to dominate certain regions of the availability-security plane. Within a region we expected the parameters to have a large impact on the performance achieved. Our investigation shows that our original insights were basically correct. However, some transitions between regions are abrupt, indicating that a large performance difference occurs for a small change in security or availability guarantees. For example, the transition from replication to information dispersal and the first transition between schemes within the region where information dispersal dominates have this property. This contradicts our original intuition that the large number of schemes would result in a selection surface that is smooth. If a system is operating near a sharp performance transition, the system requirements must be considered and an engineering decision must be made as to whether to err in favor of performance, availability, or security.

Sharp transitions tend to be localized in the back corner of the trade-off space (low availability and low security). The sharpness is due to the data redundancy of threshold schemes. Share size is calculated as $1/(m\text{-}(p\text{-}1))$, and n shares are generated by a threshold scheme. Consider a 1-1-4-threshold scheme (4-fold replication) and a 1-2-4-threshold scheme (an information dispersal scheme). By increasing m from 1 to 2, the data redundancy is halved (i.e., storage space consumption is the same as 2-fold replication). On a read, both schemes require the same amount of data (one replica versus two shares, each half the size of the replica). The abrupt performance difference on the selection surface is due to write operations, which depend on the amount of data that must be sent across the network.

Another feature of the selection surface occurs when two algorithms perform similarly; the result is a "checkerboard" on the selection surface. In such cases, it is informative to consider the resource consumption of each of the similarly performing schemes. Since the predicted difference is within the error of our model, a system designer can use secondary criteria to select schemes.

Another aspect of the space that interests us is the performance difference between algorithms. Our intuition was that limiting the set of potential algorithms would often result in poor performance. Our intuition was correct: each algorithm tends to be good for only some region of the availability-security plane. If the system being considered does not operate within the algorithm's "good" region, the system has incurred an unnecessary performance penalty. Indeed, limiting the set of algorithms reduces the region of the availability-security plane in which the system can operate. Constraining an algorithm to a specific set of parameters (i.e., building a system around a single scheme) fixes the availability and security that can be offered. Moreover, unless thorough analysis of the trade-off space was done as part of the system design, it is unlikely that the scheme selected is well matched to the system being built. This is only a reasonable design decision if the system is to operate in a single region and the scheme used is on the selection surface at the region.

A replication algorithm always achieves the highest degree of availability (because only replication can handle n-1 failures) and the highest degree of security is always achieved by splitting (because only splitting can handle n-1 compromises).

At some level of security, secret sharing dominates only the edge region that demarcates the reachable portion of the availability-security plane. This is because for an $m_1\text{-}m_1\text{-}n_1$ secret sharing scheme, there is usually a $p_2\text{-}m_2\text{-}n_2$ ramp scheme with $p_2 = m_1$, $m_2 > m_1$, and $n_2 > n_1$ which provides, by definition, the same security guarantee, provides adequate availability, but has much better performance. The reason for this is that the space savings of ramp schemes with $m > p$ result in fewer computations being required, and thus better performance (i.e., since shares are smaller in size, fewer calculations are required to generate each share). Thus, if secret sharing is the only algorithm that offers the security and availability required, increasing the number of storage nodes, (i.e., the maximum value n may take), will produce a better performing ramp scheme (>2× improvement) that meets the availability and security requirements. Clearly, this can only be done if there is not a hard design constraint on the number of storage nodes in the system.

8 PASIS CONSISTENCY PROTOCOLS

8.1 Introduction

Survivable storage systems spread data redundantly across a set of decentralized storage-nodes, in an effort to ensure the availability of that data despite the failure or compromise of storage-nodes. Perhaps one of the most difficult aspects of designing a survivable storage system is predicting the faults, threats, and environments to which it will be subjected. A system can be designed pessimistically, i.e., built upon weak assumptions, though often this comes with high performance cost. Alternatively, the system can optimistically "assume away" certain environments or threats to gain performance.

This chapter describes a family of consistency protocols that exploit data versioning within storage-nodes to efficiently provide strong consistency for erasure-coded data. The protocol family covers a broad range of system model assumptions with no changes to the client-server interface, server implementations, or system structure. For each combination of key system model assumptions (crash vs. Byzantine servers, crash vs. Byzantine clients, synchronous vs. asynchronous communication, total number of failures), there is a suitable member of the protocol family. Protocol instances are distinguished by their read and write thresholds (minimum number of storage-nodes contacted to ensure correctness), read and write policies (actual number of storage-nodes contacted), and data encoding mechanisms. Weaker assumptions lead to larger thresholds, more demanding policies, and more expensive encoding mechanisms. However, for any given set of system assumptions, the protocol is reasonably efficient. The protocol scales with its requirements—it only does work necessitated by the system model.

Each protocol in the family works roughly as follows. To perform a write, clients write time-stamped fragments to at least a write threshold of storage-nodes. Storage-nodes keep all versions of fragments they are sent. To perform a read, clients fetch the latest fragment versions from a read threshold of storage-nodes. The client determines whether the fragments comprise a consistent, complete write; usually, they do. If they do not, additional fragments or historical fragments are fetched, until a consistent, complete write is observed. Only in cases of failures (storage-node or client) or read-write concurrency is additional overhead incurred to maintain consistency.

In the common case, the consistency protocol proceeds with little overhead beyond actually reading and writing data fragments. More specifically, the protocol is efficient in three ways. First, by using *m*-of-*n* erasure codes (i.e., *m*-of-*n* fragments are needed to reconstruct the data), a decentralized storage system can tolerate multiple failures with much less network bandwidth (and storage space) than with replication [Strunk00, Wylie00]. Second, by matching the value of *m* to the read threshold size, no extra communication is required for consistency in the common case. A client crash during a write operation, a misbehaving server, and read-write concurrency can introduce protocol overhead. The first two should be very rare. As well, most studies of distributed storage systems (e.g., [Baker91, Deepinder94, Kistler92, Noble94]) indicate that minimal writer-writer and writer-reader sharing occurs (usually well under 1% of operations). Third, the execution of the protocol falls on the clients, leaving storage-nodes to the servicing of simple read and write requests. This results in scalability gains by following the well-known scalability principal of shifting work from servers to clients [Howard88]. Clients are responsible for encoding and decoding data, detecting potential consistency problems, and resolving them.

This chapter is divided into two parts: (i) a partial development of the consistency protocol family and (ii) an investigation of family members' performance behavior in a 20 node cluster.

Our partial development of the protocol family focuses on an instance suitable for use in an asynchronous system that tolerates the Byzantine failure of storage-nodes and clients. In this very weak system model, the protocol offers strong consistency semantics, namely a variant of linearizability in which read operations are permitted to abort (in presumably uncommon circumstances). We identify various other mem-

bers of the protocol family that eliminate read aborts, are better suited for stronger system models (e.g., synchrony and non-Byzantine failures), and that offer other features.

The second part of the paper investigates the performance of representative members of the protocol family. The response time of a mixed workload is shown to scale nearly linearly up to as many as 20 storage-nodes for three of the four selected protocols. The throughput that a 7 storage-node configuration can service is shown to scale nearly linearly as clients are added to the system until the network begins to saturate. The impact of read-write concurrency on system performance is shown to be nominal for a highly concurrent workload.

The initial work on this protocol was described in a CMU/SCS technical report, "Decentralized Storage Consistency via Versioning Servers" [Goodson02], and the family is described in a second technical report, "Efficient Consistency for Erasure-coded Data via Versioning Servers" [Goodson03a]. These new protocols have been integrated into the PASIS prototype.

8.2 Background

Figure 8-1 illustrates the abstract architecture of a fault-tolerant, or survivable, distributed storage system. To write a data-item *D*, Client A issues write requests to multiple storage-nodes who host the data-item. To read *D*, Client B issues read requests to an overlapping subset of storage-nodes. This basic scheme allows readers and writers to successfully access data-items even when subsets of the storage-nodes have failed. To provide reasonable storage semantics, however, the system must guarantee that readers see consistent answers. For example, assuming no intervening writes, two successful reads of *D* should produce the same answer (the most recent write) independent of which subset of storage-nodes is contacted.

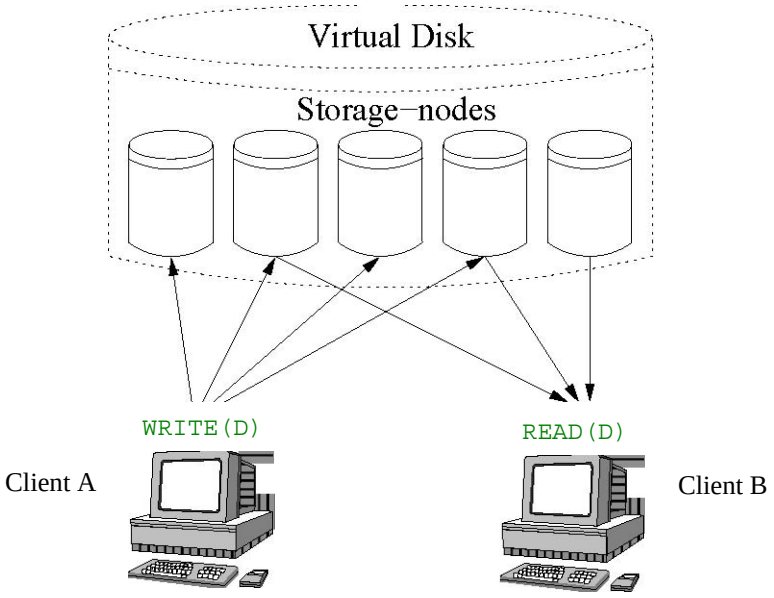


Figure 8-1. High-level architecture for survivable storage. Spreading redundant data across independent storage-nodes increases the likelihood of data surviving failures of storage nodes. Clients update multiple servers to complete a write and (usually) readers fetch information from multiple servers to complete a read.

8.2.1 Decentralized Storage

A common data distribution scheme used in decentralized storage systems is data replication. That is, a writer stores a replica of the new data-item value at each storage-node to which it sends a write request. Since each storage-node has a complete instance of the data-item, the main difficulty is identifying and retaining the most recent instance. It is often necessary for a reader to contact multiple storage-nodes to ensure that it sees the most recent instance.

Some distributed storage systems spread data among storage-nodes more space-efficiently, using erasure coding or even simple striping. With striping, a data-item is divided into fragments, all of which are needed to reconstruct the entire data-item. With erasure coding, a data-item is encoded into a set of fragments such that any sufficient subset allows reconstruction. The PASIS project [Wylie01, Wylie00] is a distributed storage system that uses erasure coding.

A challenge that must be confronted in the design of decentralized storage systems is that of partially completed write operations. Write operations in progress and incomplete write operations by clients that crash are both instances of partially completed write operations. A common approach to dealing with partial writes in non-Byzantine-tolerant systems is two-phase commit [Gray78]. This works for both replication and erasure coding, but adds a round-trip for every write. The partial write problem can also be addressed via "repair," which involves a client or storage-node distributing a partially written value to storage-nodes that have not seen it. This is only an option for systems using erasure codes when partial writes provide enough information to reconstruct the original data.

Amiri et al. [Amiri00, Amiri99] use a "stripe map" to communicate to storage-nodes the set of other storage-nodes that host related stripe units. In the Palladio project [Golding99], stripe maps enables the masking of partial writes by clients. Since client writes employ two-phase commit, storage-nodes can detect a failure of the client using a timeout. Upon client failure detection, the stripe map enables the storage-nodes to perform a reconciliation protocol that results in either the write being completed (repaired) or aborted (deleted at all nodes involved). In our protocol, when repair is allowed, it is initiated by clients and made possible by retaining versions on the storage-nodes. If repair is not allowed, each subsequent read operation requires the client to resolve the partial write operation.

8.2.2 Byzantine Fault-tolerance

Most systems that tolerate Byzantine client and server failures use Byzantine fault-tolerant agreement to maintain replicated state machines [Schneider90]. Our versioning-based protocols are enabled by the fact that storage actions (read and write) are simpler than arbitrary state machines. An alternative to replicated state machines are Byzantine quorum systems [Malkhi98], of which our protocols can be viewed as employing a particular type.

8.2.3 Consistency Semantics

Our primary target consistency semantics (linearizability with read aborts [Herlihy90]) have been studied previously. Notably, Pierce [Tumblin01] presents a protocol implementing these semantics in a decentralized storage system using replication. Our protocols go beyond this work by accommodating erasure-coded data and providing greater efficiency: e.g., our common case read operation is a single round. Initial work on our protocol is described in [Goodson02].

Versioning storage-nodes in our protocol provide capabilities similar to "listeners" in the recent work of Martin, et al. [Martin02]. The listeners protocol guarantees linearizability in a decentralized storage system. A read operation establishes a connection with a storage-node. The storage-node sends the current data-item value to the client. As well, the storage-node sends updates it receives back to the client, until the client terminates the connection. Thus, a reader may be sent multiple versions of a data item. In our protocol, readers look backward in time via the versioning storage-nodes, rather than listening into the

future. Looking back in time is more message-efficient in the common case, yields a lighter-weight server implementation, and does not necessitate repair to deal with client failures.

8.3 System Model

The infrastructure of the PASIS storage system is described in terms of *clients* and *storage-nodes*. There are *N* storage-nodes and an arbitrary number of clients in the system.

We say that a client or storage-node is *correct* in an execution if it satisfies its specification throughout the execution. A client or storage-node that deviates from its specification is said to *fail*. We assume a hybrid failure model for storage-nodes. Up to *t* storage-nodes may fail, $b \leq t$ of which may be Byzantine faults; the remainder are assumed to crash benignly. A client or storage node that does not exhibit a Byzantine failure (it is either correct or crashes) is said to be *benign*. We assume that Byzantine clients and storage-nodes are computationally bounded so that we can employ cryptographic primitives (e.g., cryptographic hash functions and encryption).

We assume that communication between clients and storage-nodes is point-to-point, reliable, and authenticated: A correct storage-node (client) receives a message from a correct client (storage-node) if and only if that client (storage-node) sent it. When convenient, we will represent communication using `SEND` and `RECEIVE` primitives. A message `RECEIVED` bears an identifier of client/storage-node from which it was received.

We consider both synchronous and asynchronous models. In an asynchronous system, we make no assumptions about message transmission delays or the execution rates of clients or storage-nodes. In contrast, in a synchronous system, there are known bounds on message transmission delays between correct clients/storage-nodes and their execution rates. As well, in the synchronous model, we assume that clients and storage-nodes have loosely synchronized clocks (i.e., in the synchronous model, all clocks are synchronized to within some constant, τ, of the same value). Protocols to achieve approximate clock synchronization in today's networks are well known, inexpensive, and widely deployed [Mills95, Mills92]. Alternately, a distinct synchronous communication path can be assumed for clock synchronization (e.g., GPS).

There are two types of *operations* in the protocol–*read operations* and *write operations*–both of which operate on *data-items*. Clients perform read/write operations that issue multiple read/write *requests* to storage-nodes. A read/write request operates on a *data-fragment*. A data-item is *encoded* into data-fragments. Requests are *executed* by storage-nodes; a correct storage-node that executes a write request is said to *host* that write operation.

Clients encode data-items in an erasure-tolerant manner; thus the distinction between data-item and data-fragment. We only consider threshold erasure codes in which any *m* of the *n* encoded data-fragments can decode the data-item. Examples of such codes are replication, Reed-Solomon codes [Berlekamp68], secret sharing [Shamir79], RAID 3/4/5/6 [Patterson88], information dispersal (IDA) [Rabin89], short secret sharing [Krawczyk94], and "tornado" codes [Luby01].

Storage-nodes provide fine-grained versioning, meaning that a correct storage-node hosts a version of the data-fragment for each write request it executes. Storage-nodes offer interfaces to write a data-fragment at a specific logical time, to query the greatest logical time of a hosted data-fragment, to read the hosted data-fragment with the greatest logical time, and, to read the hosted data-fragment with the greatest logical time at or before some logical time.

8.4 Consistency Protocol

We have developed a family of consistency protocols that efficiently support erasure-coded data-items by taking advantage of versioning storage-nodes. Members of the protocol family are differentiated based on the system model assumed (asynchronous or synchronous, type of storage-node and client failure), the

specified protocol thresholds (write and read thresholds), the specified write and read policy, additional encode mechanisms employed, and whether repair is performed. We sketch the protocol at a high level, and then give more details about the asynchronous protocol under the hybrid failure model. Other members of the protocol family are reductions or slight modifications of the asynchronous consistency protocol. In Section 8.4.4, we discuss other protocols from the same family as the asynchronous protocol.

At a high level, the protocol proceeds as follows. Logical timestamps are used to totally order all write operations and to identify write requests from the same write operation across storage-nodes. For each write, a logical timestamp is constructed that is guaranteed to be unique and greater than that of the *latest complete write* (the complete write with the highest timestamp).

To perform a read operation, clients issue read requests to a set of storage-nodes. Once at least a read threshold of storage-nodes reply, the client identifies the *candidate*, which is the data-item version returned with the greatest logical timestamp. The set of read responses that share the timestamp of the candidate are the *candidate set*. The read operation *classifies* the candidate as complete, partial or unclassifiable. If the candidate is classified as complete, then the read operation is complete; the value of the candidate is returned. If it is classified as *partial* (i.e., not complete), the candidate is discarded, a new candidate is identified, previous data-item versions are requested, and classification begins anew; this sequence may be repeated. If information has been solicited from all possible storage-nodes and the candidate remains unclassifiable, the read operation aborts.

```
READ():
1:  r:=MAX[R_min,W_min]
2:  StorageNodeSet:={1,…,N}
3:  ResponseSet:=DO_READ(StorageNodeSet,r,*)
4:  loop
5:     ⟨CandidateSet,LT_C⟩:=CHOOSE_CANDIDATE(ResponseSet)
6:     if (|CandidateSet| ≥ W_min) then
7:           /* Classify candidate as complete */
8:           Data:=DECODE(CandidateSet)
9:           RETURN(Data)
10:    else if (|CandidateSet| + (N - |ResponseSet|) < W_min – b) then
11:          /* Candidate is partial, determine new candidate */
12:          ResponseSet:=ResponseSet - CandidateSet
13:          ⟨CandidateSet,LT_C⟩:=CHOOSE_CANDIDATE(ResponseSet)
14:          ResponseSet:=ResponseSet ∪
                   DO_READ(StorageNodeSet – ResponseSet,
                            r - |ResponseSet|,LT_C)
15:    else if (r < R) then
16:          /* Candidate is unclassifiable, must read more */
17:          INCREMENT[r]
18:          ResponseSet:=ResponseSet ∪
                   DO_READ(StorageNodeSet – ResponseSet,
                            r - |ResponseSet|,LT_C)
19:    else
20:          /* Candidate unclassifiable and cannot read more */
21:          RETURN(⊥)
22:    end if
23: end loop
```

```
WRITE(Data):
1:  LT:=GET_TIME()
2:  LT:=MAKE_TIMESTAMP(LT)
3:  {D_1,…,D_N}:=ENCODE(Data)
4:  for all StorageNode ∈ {1,…,N} do
5:     SEND(StorageNode,WRITE_REQUEST,LT,D_i)
6:  end for
7:  repeat
8:     ResponseSet:=ResponseSet ∪
            RECEIVE(StorageNode,WRITE_RESPONSE)
9:  until (|ResponseSet| == W)
```

```
DO_READ(ReadSet,ReturnCount,LT)
1:  for all StorageNode ∈ ReadSet do
2:     SEND(StorageNode,READ_REQUEST,LT)
3:  end for
4:  repeat
5:     ResponseSet:=ResponseSet ∪
6:  until (|ResponsSet| == ReturnCount)
7:  RETURN(ResponseSet)
```

Figure 8-2. Asynchronous consistency protocol pseudo-code.

8.4.1 Asynchronous Protocol

We describe the asynchronous protocol in detail to give an intuition about the properties of the protocol, and the necessary conditions to achieve the properties. Pseudo-code for the asynchronous protocol is given in Figure 8-2. Terms used in the pseudo-code are explained in this section.

The write threshold, W_{min}, defines a complete write operation: a write operation is *complete* the moment $W_{min}-b$ benign storage-nodes have executed write requests. Intuitively, a consistency protocol must ensure that a write operation, once complete, replaces the previous value. The read threshold, R_{min}, ensures that the protocol provides such consistency: a read operation must have at least R_{min} read responses before identifying the candidate. Essentially, the write and read thresholds establish threshold-quorums that intersect at some number of correct storage-nodes.

Beyond ensuring consistency, the protocol should enable complete writes to be classified as such. To this end, correct clients implement write and read policies. Correct clients send write requests to all N storage-nodes; they wait for $W \geq W_{min}$ write responses before continuing. The value of W depends on the system model. For example, $W \leq N-t$ in an asynchronous system, since a client cannot wait for more responses: t storage-nodes could never respond. Correct clients consider read responses from up to $R \geq R_{min}$ storage-nodes before aborting due to an unclassifiable candidate.

The major task of the read operation is to identify, and then classify, a candidate. To classify a candidate as complete, read requests from $W_{min}-b$ correct storage-nodes that host the candidate must be collected. Two factors complicate classification: Byzantine storage-nodes and incomplete information. Up to b responses to R read requests can be arbitrary. As such, at least W_{min} responses supporting a candidate are required to classify the candidate as complete. To be safe, all storage-nodes for which there is no information must be assumed to not host the candidate. Line 6 of `READ()` tests the above condition to classify the candidate under consideration. Line 1 of `READ()` ensures that a read operation may complete with a single round of communication.

To classify a candidate as partial, the read operation must observe that it is impossible for $W_{min}-b$ or more correct storage-nodes to host the candidate. To be safe, all storage-nodes for which there is no information must be assumed to host the write. Line 10 of `READ()` tests the partial classification condition.

In some cases, it is impossible for the read operation to classify the candidate as complete or partial. Interestingly, both partial and complete writes can be unclassifiable. Line 15 of `READ()` determines whether the read policy (R) will allow more read requests, or if the operation must abort

On line 1 of `WRITE()`, the function `GET_TIME()` is called. The current logical time of the data-item is determined by considering time query responses from at least R_{min} storage-nodes. The implementation of `GET_TIME()` is very similar to the function `DO_READ()`, except the *ResponseSet* consists solely of logical timestamps. The high bits of a logical timestamp are the *data-item time*. To make a timestamp, the client increments the data-item time and appends its client identifier and request identifier. The low bits of the timestamp distinguish write operations issued at the same logical data-item time.

The protocol supports erasure-coded data. The functions `ENCODE()` on line 3 of `WRITE()` erasure codes the data-item into N data-fragments. Conversely, function `DECODE()` on line 8 decodes m data-fragments, returning the data-item. Mechanisms that provide confidentiality and integrity guarantees can be employed in concert with erasure codes. Such mechanisms and the properties they provide are discussed more fully in Section 8.4.3.

8.4.2 Asynchronous Protocol Properties

We developed the protocol to have two major properties. First, it ensures the consistency of stored data-items. Second, it ensures that if all clients are correct, write and read operations that follow the write and read policies will complete. In this section, we develop constraints on W_{min}, R_{min}, W , R, and N for the asynchronous protocol, which are sufficient to achieve the desired properties. Formal statements of the desired properties and proofs that they are achieved will be provided in an extended version of this paper.

Linearizability with read aborts: Operations are *linearizable* if their return values are consistent with an execution in which each operation is performed instantaneously at a distinct point in time between its invocation and completion [Herlihy90]. *Linearizability with read aborts* restricts the definition of lineariza-

bility to write operations and read operations that return a value. As such, read operations that abort are excluded from the schedule of linearized operations. Linearizability with read aborts is similar to Pierce's "pseudoatomic consistency" [Tumblin01].

To achieve linearizability with read aborts, the protocol must ensure that a read operation will notice the latest complete write operation as a candidate (assuming the read operation does not abort). Therefore, it is necessary that a read operation and a write operation "intersect" at least one correct storage-node:

$$b + N < W_{min} + R_{min}. \tag{8.1}$$

Operations can terminate: A necessary condition for liveness in the asynchronous model is that,

$$W_t R \leq N - t. \tag{8.2}$$

Without this constraint, write and read operations could await responses forever, thus never completing. Constraints (8.1) and (8.2) yield lower bounds on W_{min}, R_{min}, and N (remember, $W_{min} \leq W$, and $R_{min} \leq R$):

$$t + b < W_{min}; \tag{8.3}$$

$$t + b < R_{min}; \tag{8.4}$$

$$2t + b < N. \tag{8.5}$$

Constraints (8.3) and (8.4) follow from substituting (8.2) into (8.1). Constraint (8.5) follows from the resulting lower bounds on W_{min} and R_{min}.

Read operations can return a value: Read operations must be able to return a value despite the presence of Byzantine storage-nodes in the system. If Byzantine storage-nodes can always fabricate write requests that a read operation deems unclassifiable, then all read operations can be forced to abort. To ensure that Byzantine storage-nodes cannot always fabricate an unclassifiable candidate, a candidate set of size b must be classifiable as partial. To classify such a candidate as partial, it is necessary that,

$$B + (N - R) < W_{min} - b. \tag{8.6}$$

Constraint (8.6) creates a relation between the read policy, R, and the definition of a complete write (W_{min}). The less "aggressive" the read policy, the larger W_{min} must be to ensure that Byzantine storage-nodes cannot force all read operations to abort.

Assuming a read policy of $R=N - t - \kappa$, where κ determines the exact read policy R, constraint (8.6) becomes,

$$B + (N - (N - t - \kappa)) < W_{min} - b_t$$

$$t + 2b + \kappa < W_{min}. \tag{8.7}$$

Constraint (8.7) is more restrictive (on W_{min}) than our previous lower bound, $t + b < W_{min}$ (cf. (8.3)). The most aggressive read policy, $\kappa = 0$, yields $t + 2b < W_{min}$.

Correct write operations are classi able: Write operations by correct clients should be classifiable as complete. As such, the minimum intersection of write requests and read requests at correct storage-nodes guaranteed by the write and read policies must ensure sufficient information to classify the candidate as complete:

$$W + R - N - b \geq W_{min}. \tag{8.8}$$

The term $W + R - N$ is the minimum intersection of write and read requests; up to b of the storage-nodes in the intersection may be Byzantine. Assuming the most aggressive write and read policies ($W = R = N - t$), constraint (8.8) becomes:

$$(N - t) + (N - t) - N - b \geq W_{min};$$

$$N - 2t - b \geq W_{min} \tag{8.9}$$

Which implies, $W_{min} + 2t + b \leq N$, a more restrictive lower bound on N, than constraint (8.5).

Complete write operations are decodable: A complete write must be sufficient to decode a data-item. To achieve this property,

$$m \leq W_{min} - b. \tag{8.10}$$

It is safe to let $m = 1$ (i.e., replication). But, we are interested in the space-efficiency offered by erasure codes. As such, the upper bound on m is of interest, since the sizes of data-fragments are inversely proportional to m.

Given the above constraints, there is still a large space from which to select values for W_{min}, R_{min}, W, R, and N. To limit the scope of this investigation, we focus on the smallest values of parameters W_{min}, R_{min}, and N that provide the desired properties. Assuming the most aggressive read and write policies, $W = R = N - t$, constraints (8.1), (8.7), and (8.9), demand that,

$$\begin{aligned} T + 2b &< W_{min} \\ W_{min} + 2t + b &\leq N \\ W_{min} + R_{min} &> N + b \end{aligned} \tag{8.11}$$

Thus, we focus on the threshold values $W_{min} = t + 2b + 1$, $R_{min} = 2t + 2b + 1$, and $N = 3t + 3b + 1$, which meet all of the constraints.

8.4.3 Data Encoding

The implementation of `ENCODE()` and `DECODE()` are very important to the properties achieved by the protocol in Figure 8-2. Here, we describe three mechanisms that can be incorporated into their implementation, and the properties that these mechanisms provide.

Cross checksums: Cross checksums are used to detect if Byzantine storage-nodes corrupt data-fragments that they host. After erasure coding, a cryptographic hash of each data-fragment is computed, and the set of N hashes is concatenated to form the *cross checksum* of the data-fragments. The cross checksum is then appended to each data-fragment. A read operation uses the cross checksum to validate the integrity of data-fragments: each data-fragment that does not match its portion of the cross checksum is discarded. More than b matching instances of the cross checksum must be observed before it is used to validate data-fragments. Cross checksums of erasure-coded data were proposed by Gong [Gong89]. Krawczyk [Krawczyk93] extended cross checksums to make use of error-correcting codes; the space-efficiency of Krawczyk's *distributed fingerprints* comes at a cost in computation and complexity.

Validated cross checksums: In system models that admit Byzantine clients, cross checksums can be extended to detect Byzantine clients who write data-fragments that are not consistent (i.e., where different sets of m data-fragments reconstruct a different data-item). Given m data-fragments that are consistent with a cross checksum received from more than b storage nodes, the reading client regenerates the remaining $N - m$ data-fragments. If any of these $N - m$ generated data-fragments are inconsistent with the cross checksum, then decoding fails.

Note that this procedure requires that any m data-fragments uniquely determine the other $N - m$. This holds for the erasure-coding schemes that we employ.

When employing this mechanism, there are two options for where to place the cross checksum. In the first, the cross checksum is embedded as the low order bits of the logical timestamp of the write operation; we call this construction a *self-validating timestamp*. The second option is to leave the timestamp construction unchanged, and to append the cross checksum to each data-fragment as above. In this case, it is necessary to impose the constraint $2W_{min} > N + b$ to prevent a Byzantine client from completing writes of two distinct data-items with the same logical time.

Short secret sharing: Short secret sharing [Krawczyk94] provides data confidentiality so long as fewer than m data-fragments are observed. Thus, if $m > b$, Byzantine storage-nodes are prevented (computationally) from leaking information about the contents of data-items. Information about access patterns and data-item size can still be leaked, of course.

8.4.4 Other Members of the Protocol Family

There are several branches of the protocol family that yield interesting protocols: achieved properties can be weakened, stronger assumptions can be made, and repair can be leveraged. The asynchronous protocol described, achieves strong liveness guarantees. Granting Byzantine storage-nodes the power to prevent read operations from completing creates a larger set of valid protocol thresholds—these are attractive if the lower bounds on W_{min}, R_{min}, and N are of concern. We do not experiment with protocols that do not achieve the above properties in this work.

The assumption of synchrony defines another branch of the protocol family tree. This assumption establishes the ability to have loosely synchronized clocks and detect crashed storage-nodes via timeouts. The former is used to make `GET_TIME()` a local operation (cf. line 1 of the write operation). In this case, global time is used as the high bits of the logical timestamp. This reduces a write operation to a single round of communication with storage-nodes. The definition of operation duration is extended by the clock skew of the clock synchronization protocol. Many have exploited the fact that messaging overhead and round-trip counts can be reduced with loosely synchronized clocks (e.g., Adya et. al [Adya95, Adya97]).

Since crashed storage-nodes can be detected in the synchronous model, $W = N$ and $R = N$ are allowable write and read policies. Consequently, the lower bounds on protocol thresholds in the synchronous model are lower than in the asynchronous. As well, responses from all correct storage-nodes can be solicited; the lower bound on the range of candidate sets that are unclassifiable can be reduced to b (from $t + b$ in the asynchronous model).

Introducing repair into the protocol enables it to provide linearizability rather than linearizability with read aborts. To accomplish this, all unclassifiable candidates must be repairable. This constraint reduces the upper bound on m, making the protocol less space-efficient. However, the lower bound on N is also reduced; with repair, Byzantine storage-nodes need not be prevented from making complete write operations by correct clients unclassifiable. We include repairable protocols for the asynchronous and synchronous models in our experiments.

The members of the protocol family that we evaluate in Section 8.6 are listed in Table 8-1. The values of N and W_{min} in the asynchronous and asynchronous with repair systems are lower bounds. For synchronous systems, we experiment with thresholds we know to be correct (i.e., that achieve the desired properties), but we believe are not lower bounds (i.e., we have not completed the proof sketch for lower bounds yet, but our intuition is that N can be much lower).

Protocol	W_{min}	R_{min}	N	m
Asynch.	$t + 2b + 1$	$2t + 2b + 1$	$3t + 3b + 1$	$t + b + 1$
Synch.	$t + b + 1$	$t + b + 1$	$2t + b + 1$	$t + 1$
Asynch.+Repair	$t + 2b + 1$	$t + b + 1$	$2t + 2b + 1$	$b + 1$
Synch.+Repair	$t + b + 1$	$t + b + 1$	$2t + b + 1$	$b + 1$

Table 8-1. Example protocol instances. The protocol thresholds are listed for the set of protocol instances evaluated in Section 6. All protocol instances listed employ the most aggressive write and read policy possible (e.g., $W = R = N - t$ in the asynchronous model).

8.4.5 Byzantine Clients

If an authorized client is Byzantine, there is little a storage system can do to prevent the client from corrupting data. Such a client can delete data or modify it arbitrarily. The best a storage system can do is to provide mechanisms that facilitate the detection of incorrect clients and the recovery from Byzantine client actions.

Storage-based intrusion detection could assist in the detection of malicious clients [Pennington02, Strunk02]. As well, since the consistency protocol makes use of fine-grained versioning storage-nodes (i.e., self-securing storage [Strunk00]), recovery and diagnosis from detected storage intrusions is possible. Maliciously deleted data can be recovered, and arbitrarily modified data can be rolled back to its pre-intrusion state.

Encoding mechanisms can limit the power of Byzantine clients; however, they still have some power. Worse, collusion with Byzantine storage-nodes enables a Byzantine client to perform a write operation that is guaranteed to be unclassifiable (without repair, such operations result in read aborts). A Byzantine client can perform a large number of intentional partial write operations such that a subsequent read operation must issue many read requests to complete. Finally, a Byzantine client can perform write operations that result in integrity faults. In some situations, such faults cannot be masked, though they are always detected.

In a synchronous system, in which clients use synchronized clocks, Byzantine clients can perform write operations "in the future." Synchronizing storage-nodes clocks with client clocks would enable storage nodes to bound how far into the future write operations can be performed (e.g., to 2τ).

8.5 Prototype Implementation

This section describes a survivable block-store based on erasure-coding schemes, versioning storage nodes, and the consistency protocol described in Section 4. The client software can be configured to support different system model assumptions and failure tolerances. A given configuration affects the protocol thresholds and particular erasure-coding scheme utilized.

The system consists of clients and storage-nodes. The client module provides a block-level interface to higher-level software, and uses a simple read-write RPC interface to communicate with storage-nodes. The client module is responsible for encoding and decoding data-item blocks to and from data-fragments, and for the execution of the consistency protocol. The storage-nodes are responsible for storing data-item fragments and their versions.

8.5.1 Storage-nodes

Storage-nodes provide storage and retrieval of data-fragment versions for clients. Table 8-2 shows the interface exported by a storage-node.

We use the Comprehensive Versioning File System [Soules03] as the versioning storage-nodes.

RPC Call	Description
`Read`	Read a block at/or before a logical timestamp (or latest version)
`Write`	Write a block with logical timestamp
`GetLTime`	Get logical data-item time of a block

Table 8-2. Remote Procedure Call List

Time: Storage-nodes provide an interface for retrieving the logical time of a data-item, but do not enforce its use. In the synchronous timing model, recall, clients are correct in using their local clocks as logical timestamps. To break ties, the client ID and request ID are concatenated to the logical timestamp.

Writes: In addition to data, each write request contains a *linkage record* and, optionally, data-item cross checksums. The cross checksums are stored with the data-fragment. The linkage record consists of the addresses of all the storage-nodes in the set of N for a specific data-item. Since there is no create call, the linkage information must be transmitted on each write request. Linkage records enable storage-nodes to perform decentralized garbage collection of old versions. Linkage records are similar to the "stripe maps" used in Cheops [Amiri00] and Palladio [Golding99]. Indeed, the linkage record structure is introduced by Amiri and Golding [Amiri99].

Reads: By default, a read request returns the data of the most current data-fragment version, as determined by the logical timestamps that accompany the write requests. To improve performance, read requests may also return a limited version history of the data-fragment being requested (with no corresponding data). Each history entry consists of a `(client logical timestamp, cookie)` tuple. This version history information allows clients to classify earlier writes without extra requests to storage-nodes. The cookie is an opaque data handle that can be returned to the storage-node on a subsequent read to efficiently request the data version that matches a given history entry.

Data versioning: The storage-node implementation uses a log-structured data organization to reduce the cost of data versioning. Like previous researchers [Strunk00], our experiences indicate that retaining every version and performing local garbage collection come with minimal performance cost (a few percent). Also, previous research [Quinlan02, Strunk00] indicates that the space required to retain multi-day version histories is feasible.

Garbage Collection: Pruning old versions, or garbage collection, is necessary to prevent capacity exhaustion of the backend storage-nodes. A storage-node in isolation, by the very nature of the protocol, cannot determine what local data-fragment versions are safe to garbage-collect. This is because write completeness is a property of a set of storage-nodes, and not of a single storage-node. An individual storage-node can garbage-collect a data-fragment version if there exists a later data-fragment version that is part of a complete write.

Storage-nodes are able to classify writes by executing the consistency protocol in the same manner as the client. Classification requires that storage-nodes know how to contact the other N - 1 nodes hosting the write operation. Linkage records provide this information. As well, they also provide a means of access control; a storage-node can deny requests from all nodes not contained within the linkage record for a given data-fragment. In addition, implementation does not require the use of additional RPCs. Furthermore, garbage collection does not require the transfer of data, only that of history version information.

Policy issues remain regarding the frequency with which storage-nodes should perform garbage collection and the granularity at which it should be done.

8.5.2 Clients

The bulk of the consistency protocol is handled by the clients. The clients are responsible for encoding data blocks into data-fragments and storing them across a set of distributed storage-nodes. The clients are also responsible for enforcing the consistency properties through the correct classification of read requests.

8.5.2.1 Client interface and organization

In the current implementation, the client-side module is accessed through a set of library interface calls. These procedures allow applications to control the encoding scheme, the protocol threshold values (N, R, R_{min}, W, W_{min}), failure model (b, t, Byzantine clients), and timing model (synchronous or asynchronous),

as well as more routine parameters such as block size. The client protocol routines are implemented in such a way that different parameters may be specified for different sets of data-items.

Writes: The client write path is implemented as follows. First, depending on the timing model, a logical timestamp is created. In the asynchronous case, this involves sending requests to at least R_{min} storage-nodes in order to retrieve the data-item time. In the synchronous model, the client's clock is read (all clients are synchronized using NTP [Mills92]). A client specific request ID and client ID are then appended to create the logical timestamp.

Next, the data block is encoded into data-fragments using the specified scheme (see Section 8.5.2.2). A set of storage-nodes is then selected. If the write is a block overwrite, a list of storage-nodes must be passed into the write procedure, if not, a set of nodes is selected randomly from a list of well-known storage-nodes. Better procedures for selecting "good" storage-nodes are outside the scope of this paper.

Finally, write requests for each encoded data-fragment are issued to the set of N storage-nodes. Each write includes the linkage record, cross checksum, and logical timestamp. The write completes once W_{min} completions have been successfully returned. Upon completion, the linkage record is returned to the caller. The remaining N - W responses are dropped by the client, although the writes may execute at the storage-nodes.

Reads: The client read path is somewhat more complicated, since it executes the resolution portion of the consistency algorithm. The application is expected to provide the linkage record naming the set of N storage-nodes from a previous write. Read requests are issued to the first R_{min} storage-nodes, so that systematic decoding of the data can be performed if all requests are successful. Once the R_{min} requests have completed, read classification begins.

Read classification proceeds as described in the pseudo code, in Figure 8-2, with a few optimizations. Since storage-nodes return a set of data-fragment version histories, the client need issue fewer read requests for timestamps of previous versions. Classification continues until either a complete read is found or the client exhausts the set of storage-nodes it can query.

Once classification completes and a complete read has been found, decoding of the data block can be attempted. To do so, it is necessary to have data from at least m data-fragments belonging to the same logical timestamp. If the data from less than m matching data-fragments are available (due to the use of version history information), it is necessary to request additional data from storage-nodes hosting the candidate write. Once m matching data-fragments have been collected, decoding is performed.

If cross checksums are in use (to tolerate Byzantine failures), checksum validation is first performed. If checksum validation fails, the integrity of the data-item cannot be guaranteed and the implementation currently returns an integrity fault. Similarly, if validated cross checksums or self-validating timestamps fail, an integrity fault occurs. If the data block is successfully decoded, it is returned to the client.

Metadata: For simplicity and modularity of the implementation, it is assumed that the client application maintains metadata as to where data blocks are stored (i.e., the linkage records). For example, to build a file system, linkage records could be stored within directories and inodes that are themselves stored by the block store. In this case, there must be some way of determining the root of the file system. For our experiments, a static set of N storage-nodes is used, obviating the need to save individual linkage records.

8.5.2.2 Encode and decode implementation

The encode and decode functions are tuned for efficiency. Erasure coding is used when availability is the focus and short secret sharing when confidentiality (from servers) is desired. This subsection describes both, as well as cryptographic algorithms.

Erasure codes: Although we have implementations of replication and RAID (i.e., single parity block), we focus on erasure codes that can protect against more than a single or double erasure.

Our erasure code implementation stripes the data-item across the first m data-fragments. Each *stripe-fragment* consists of $1/m$ the length of the original data-item.

Stripe-fragments are used to generate the *code-fragments*. Code-fragments are created by treating the m stripe-fragments as a vector of y values for some unique x value in a Galois Field — polynomial interpolation within the Galois Field based on the stripe-fragments yields the code-fragments. Each code-fragment has a unique x value as well. We use a Galois Field of size 2^8 so that each byte in the data-item corresponds to a single y coordinate. This decision limits us to 256 unique x coordinates (i.e., $N \leq 256$).

Since the polynomial is interpolated at the x coordinate points many times, interpolation coefficients are precalculated using "Newton's Formula" [Knuth81]. Given these $m \times (N - m)$ constants, each of the $N - m$ code-fragment interpolations require m multiplications and $2m$ additions. Multiplication in the Galois Field is implemented as a lookup table (i.e., a $2^8 \times 2^8$ B = 64 KB lookup table is used). Addition in the Galois Field is just bitwise XOR. Although this implementation more closely resembles the algorithm described by Shamir for sharing secrets [Shamir79], we refer to it as information dispersal [Rabin89] because of its lack of secrecy.

Our implementation of polynomial interpolation was originally based on publicly available code for information dispersal [Dai]. We note that Hal Finney's 'secsplit.c' was acknowledged in [Dai]. We modified the source to make use of stripe-fragments and the lookup table. As well, some loops were unrolled so that addition (XOR) could occur on word boundaries, rather than byte boundaries.

Stripe-fragments make the erasure code a *systematic encoding*. If the first m data-fragments are passed into the decode function, no polynomial interpolation is required—the stripe-fragments need only be copied into the data-item buffer. Each stripe-unit passed in to decode reduces the number of fragments that must be interpolated during decode by one.

Short secret sharing: Our implementation of short secret sharing closely follows [Krawczyk94]. We use a cryptographically secure pseudo-random number generator to generate an encryption key. The key is used to encrypt the data-item buffer under AES in CBC mode. Data-fragments written to storage-nodes consist of a fragment based on the key and a fragment based on the encrypted data-item.

The key is erasure-coded using Secret Sharing [Shamir79]. In secret sharing, the data-item is treated as a data-fragment, $m - 1$ random data-fragments are generated, and $N - m + 1$ code-fragments are interpolated. The data-fragment based on the data-item is not stored (i.e., the key is not stored).

The portions of data-fragments that correspond to the encrypted data-item are generated using the erasure code technique outlined above.

To decode m data-fragments, the first portions are decoded via secret sharing to recover the key. The encrypted data-item is recovered via the normal decode technique for erasure-coded data described above. Finally, the key is used to decrypt the data-item.

Cryptographic algorithms: We use publicly available implementations of SHA1 [Dai], a cryptographically secure pseudo-random number generator (based on ANSI X9.17 Appendix C) [Dai], and AES [Gladman]. We note that Steve Reid is acknowledged as the originator of the SHA1 code [Dai].

8.6 Evaluation

This section uses the prototype system to evaluate performance characteristics of the protocol family.

8.6.1 Experimental Setup

We use a cluster of 20 machines to perform experiments. Each machine is a dual 1GHz Pentium III machine with 384 MB of memory. Each storage-node uses a 9GB Quantum Atlas 10K as the storage device. The machines are connected through a 100Mb switch. All machines run the Linux 2.4.20 SMP kernel.

Each storage-node is configured with 128 MB of data cache, and no caching is done on the clients. All experiments show results using write-back caching at the storage nodes, mimicking availability of 16 MB of non-volatile RAM. This allows us to focus on the overheads introduced by the protocol and not those introduced by the disk subsystem.

All experiments employ a similar base policy:

Clients issue writes to all N storage-nodes, but wait for only W_{min} responses.

Clients read the first R_{min} storage-nodes, where the first m data-fragments are stored, taking advantage of systematic encodings.

Clients write in a manner such that the systematically encoded data-fragments (i.e., the stripe-fragments) are randomly distributed across storage-nodes, avoiding hot spots due to the above read policy.

Clients keep a fixed number of read and write operations outstanding. Once an operation completes, a new operation is issued. Unless otherwise specified, requests are for 16 KB blocks with a 2:1 ratio of reads to writes.

Information dispersal is employed in all experiments for which $b = 0$. Short-secret sharing and cross checksums are employed in all experiments for which $b > 0$.

For the asynchronous model, clients fetch logical times from storage-nodes before issuing a write.

8.6.2 Encode and Decode Performance

The *blowup* of an erasure code indicates the proportion of network and storage capacity consumed by the encoded data. The blowup has a significant impact on the performance of systems that use erasure codes. Note, in coding theory, the inverse of the blowup is called the *rate* of a code (i.e., blowup is not a standard term from coding theory).

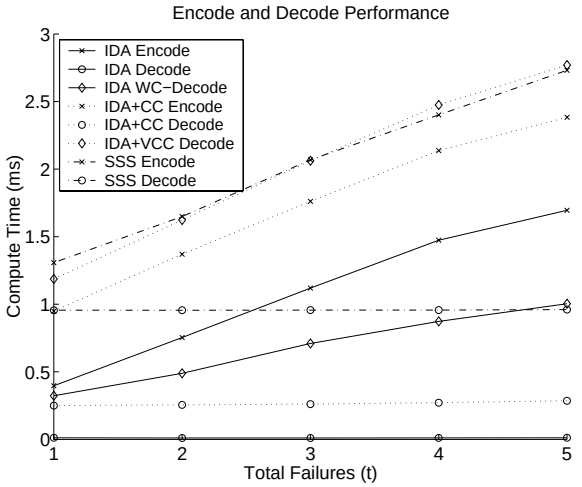


Figure 8-3. Encode and decode of 16 KB blocks. The erasure code parameters (N & m) are taken from the asynchronous model with $b = t$ (i.e., $N = 6t + 1$, $m = 2t + 1$). In the legend, IDA means information dispersal algorithm, SSS means short secret sharing, WC means worst case, CC means cross checksum, and VCC means validated cross checksum. Measurements of decode cost assume a systematic decoding using stripe fragments, except for the worst case decode measurement.

For our basic erasure code, the blowup is *N/m*. In Figure 8-3, the blowup of the erasure codes is *(6t+1)/(2t + 1).* So, for $t = 2$, $m = 5$, $N = 13$, and the blowup of the encoding is 13*/5.* Thus, a 16 KB block is encoded into 13 data-fragments, each of which is 3.2 KB in size. In total, these data-fragments consume 41.6 KB of network and storage capacity.

A more complicated formula for blowup is required for short secret sharing and cross checksums. Both mechanisms add additional overhead: the space for secret sharing the key and the space for the cross checksum respectively. The three solid lines in Figure 8-3 show performance for IDA encode and decode. The cost of IDA encode grows with *t* because *m* grows with *t*; the cost of interpolating each code byte grows with *m*.

The common-case IDA decode exploits the systematic encoding to achieve near-zero computation cost (i.e., just the cost of `memcpy`). The IDA WC-Decode line represents the "worst case" in which only code-fragments are available. The decode, therefore, requires interpolation of *m* data-fragments. IDA encode is more expensive than worst case decode because encode must interpolate $N - m > m$ data-fragments.

The three dotted lines show the cost of encode and decode when IDA is combined with cross-checksums. The IDA+CC Encode line, when compared to the IDA Encode line, shows the cost of computing SHA1 digests of the *N* data-fragments. The IDA+CC Decode line, when compared to the IDA Decode line, shows the cost of computing *m* SHA1 digests of data-fragments. The IDA+VCC Decode line, representing validated cross checksums, shows the cost of generating *N* - *m* data-fragments and taking their SHA1 digests on decode.

The two dashed lines show the cost of encoding and decoding a data-item with short secret sharing. The difference between the SSS Encode line and IDA Encode line, as well as the SSS Decode line and the IDA Decode line, is the cost of AES encryption of a 16 KB block, as well as key generation and secret sharing.

The cryptographic algorithms we use have the following performance characteristics on the testbed systems: AES 17.6 MB/s, SHA1 66.4 MB/s, and random number generation 4.6 MB/s.

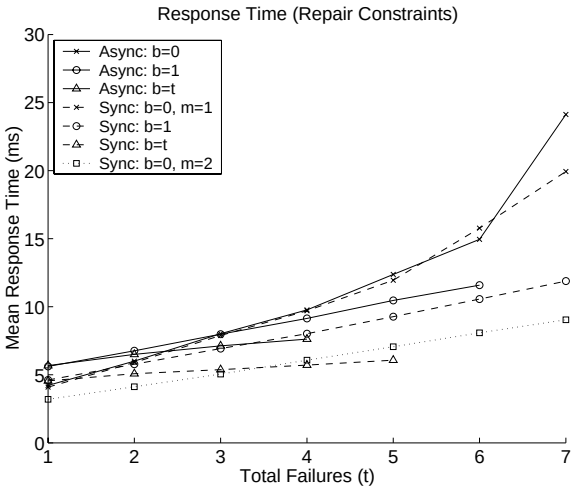


Figure 8-4. Mean response time vs. Total failures (t) (Base Constraints). Compares the mean response time of requests given different failure and timing models using non-repair thresholds. The number of storage-nodes grows with *t*, *b*, and the timing model.

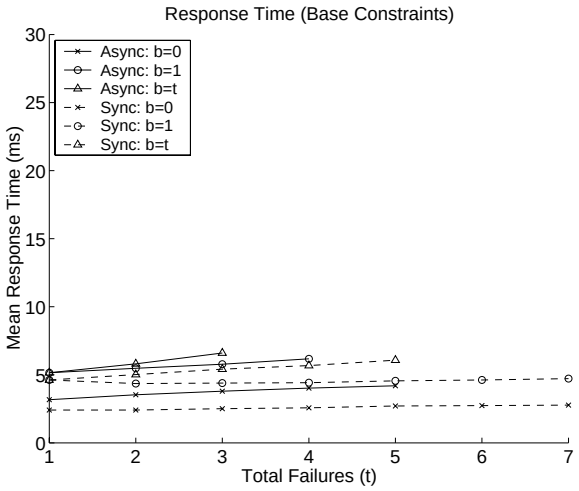


Figure 8-5. Mean response time vs. Total failures (t) (Repair Constraints). Compares the mean response time of requests given different failure and timing models using repair constraints.

8.6.3 Protocol Overheads

To evaluate the overhead of the protocol under different system models, we consider asynchronous and synchronous members of the protocol family over a range of failure tolerance. We examine the mean response time of a single request from a single client, as well as the system throughput of multiple clients with multiple requests outstanding as failure tolerance scales.

8.6.3.1 Response time

Figure 8-4 shows mean response times for several system configurations using the base constraints (i.e., those without repair in Table 8-1). Figure 8-5 shows mean response times using the repair constraints.

The mean response times are plotted as a function of t, the total number of failures the system can handle. Notice, as t increases, so does R_{min}, W_{min}, and N (see Table 8-1 for equations). Three lines are shown for each of the asynchronous and synchronous models; those that tolerate $b = 0$, $b = 1$, and $b = t$ Byzantine failures out of the t tolerated failures. To put these parameters in context, consider the asynchronous protocol with $b = t$ in Figure 8-4. The line for this protocol is only plotted until $t = 3$. At this point, there are $N = 3t + 3b + 1 = 9 + 9 + 1 = 19$ storage-nodes.

The focus of these plots is on slope of the lines. The slope shows the protocol overheads in terms of network cost, since the number of storage-nodes that are accessed is increased. The flatness of the lines shows that most network transmissions can be fully overlapped. The slope of the lines is dictated by both the number of nodes and the blowup of the encoding.

The blowup of encoding directly affects the amount of data put on the wire. The impact of this can most clearly be seen in Figure 8-5 for the $b = 0$ lines. Both original lines have $m = 1$, i.e., data-fragment size equals the original data-item size — replication is being used. This gives an increase in the amount of network traffic proportional to N. To see the effect of m on response time, compare the following lines: *Sync: b=0, m=1* and *Sync: b=0, m=2*. The m=2 line bumps R_{min}, W_{min}, N, and m up by one, reducing the data moved over the network by approximately a factor of 2. This results in a slightly lower availability, because there is one more storage-node that could fail, but much improved performance (by just over a factor of two for t=7). This is a prime example of why supporting erasure codes is desirable.

The difference in cost between *Sync: b=0* and *Async: b=0* is the sum of the cost of short secret sharing versus IDA and the extra round-trip time to fetch the logical timestamp. The $b=t$ lines have higher latencies because R_{min}, W_{min}, and N grow at a larger rate in relation to t.

8.6.3.2 Throughput

Figure 8-6 shows the throughput of a system of seven storage-nodes, in terms of requests/second, as a function of client load. Each client keeps four requests outstanding to increase the load generated. Each client accesses a distinct set of blocks, contributing to load without creating concurrent accesses to specific blocks.

As expected, throughput scales with the number of clients, until servers cannot handle more load, because of network saturation. The *b=0* configurations provides higher throughput because the blowup is smaller (i.e., the total network bandwidth consumed per operation is lower). The *Async* configurations generally saturate sooner than *Sync* configurations, because they also query servers for logical timestamp values on write requests.

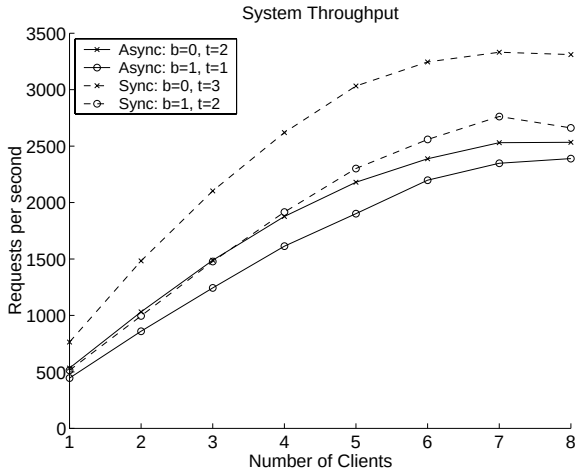


Figure 8-6. Throughput vs. Client Load. Compares the total system throughput of a mixed read/write workload as number of clients increase.

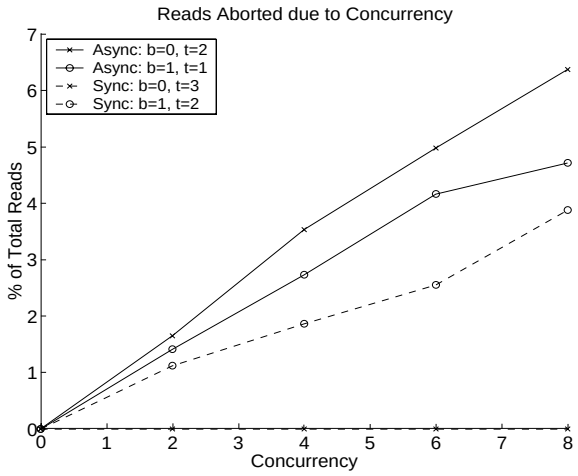


Figure 8-7. Percentage of Reads Aborting vs. Concurrency. Compares the percentage of reads that aborted (out of the total issued) due to the amount of concurrency in the system.

8.6.4 Concurrency

To measure the effect of concurrency on the system, we measure multi-client throughput when accessing overlapping block sets. The experiment makes use of four clients; each client has four operations outstanding. Each client accesses a range of eight data blocks, with no two outstanding requests going to the same block. The number of blocks in a client's block range that clients share is denoted as the concurrency level; e.g., at a concurrency of six, six blocks are shared among all clients, while two blocks are reserved exclusively for each client. Blocks are chosen from a uniform random distribution, with a read/write ratio of 50%.

At the highest concurrency level (i.e., all eight blocks are in contention by all clients), we observed neither significant drops in the bandwidth nor significant increases in mean response time. However, the standard deviations of bandwidth and response time did rise slightly. Instead of plotting response time or throughput, the discussion focuses on the subtle interactions between the protocol and concurrency.

At the highest concurrency level, we observed that the initial candidate is classified as complete 89% of the time. In the remaining 11% of the cases, more information is required. The request latencies for the latter cases increased slightly due to the extra round trips necessary to complete classification. Since this occurs so seldom, and since round trip times are fairly small, the effect on mean response time is minimal.

The other case worthy of more detail is that of aborted reads due to read/write concurrency.

8.6.4.1 Aborted reads

A write operation that is in flight concurrently to a read operation may be observed by the read as partial (although it may later complete). If a read observes a partial write that is unclassifiable (according to the classification constraints), it has to abort. This is even true in the absence of failures. We call these types of aborts, *false aborts*. This section examines the effect of concurrency on false aborts.

Figure 8-7 shows the percentage of read operations that abort due to read-write concurrency as a function of system concurrency. The *Sync:b=0* configuration never aborts due to read-write concurrency in the case of no storage-node failures. This is because it is always able to obtain perfect information about the system. All writes are classifiable as either partial or as complete.

The other three configurations, *Async: b=0, b=1, Sync: b=1*, operate under imperfect information. Imperfect information comes from either having to tolerate liars (Byzantine nodes) or from not being able to

wait for communication from all storage-nodes (asynchronous model). The number of read aborts in each of the above three configurations is proportional to the amount of information that is unavailable.

The protocol constraints preclude the *Sync: b=1* configuration from believing one response given it hears back from N. The *Async: b=0, t=2* configuration is precluded from communicating with t=2 nodes. Finally the *Async: b=1, t=1* configuration is precluded from communicating with t=1 nodes and believing an additional b=1 nodes. The difference in magnitude between the two *Async* lines comes from the ratio of $W_{\min}$ to $R_{\min}$. In the *Async: b=1, t=1* configuration, there are $\binom{6}{4}$ combinations that a read could observe, while in the *Async: b=0, t=2* configuration there are only $\binom{5}{3}$ such combinations.

8.6.4.2 Retries

Since false aborts occur due to read/write concurrency (not failures), retrying reads may lead to the determination of a complete value as the system makes progress. Even the case of continuing high concurrency, one can reduce the probability of observing a complete write by varying the retry attempts. We experimentally ran a configuration that retried aborted reads. With a maximum of three retries, we observed one abort in ≈16000 sample requests. Continuing retries whenever new responses are observed can reduce this further.

9 SELF-SECURING STORAGE & VERSIONING FILE SYSTEM COMPONENTS

9.1 Introduction

Despite the best efforts of system designers and implementers, it has proven difficult to prevent computer security breaches. This fact is of growing importance as organizations find themselves increasingly dependent on wide-area networking (providing more potential sources of intrusions) and computer-maintained information (raising the significance of potential damage). A successful intruder can obtain the rights and identity of a legitimate user or administrator. With these rights, it is possible to disrupt the system by accessing, modifying, or destroying critical data.

Even after an intrusion has been detected and terminated, system administrators still face two difficult tasks: determining the damage caused by the intrusion and restoring the system to a safe state. Damage includes compromised secrets, creation of back doors and Trojan horses, and tainting of stored data. Detecting each of these is made difficult by crafty intruders who understand how to scrub audit logs and disrupt automated tamper detection systems. System restoration involves identifying a clean backup (i.e., one created prior to the intrusion), reinitializing the system, and restoring information from the backup. Such restoration often requires a significant amount of time, reduces the availability of the original system, and frequently causes loss of data created between the safe backup and the intrusion. Self-securing storage offers a partial solution to these problems by preventing intruders from undetectably tampering with or permanently deleting stored data. Since intruders can take the identity of real users and even the host OS, any resource controlled by the operating system is vulnerable, including the raw storage. Rather than acting as slaves to host OSes, self-securing storage devices view them, and their users, as questionable entities for which they work. These self-contained, self-controlled devices internally version all data and audit all requests for a guaranteed amount of time (e.g., a week or a month), thus providing system administrators time to detect intrusions. For intrusions detected within this window, all of the version and audit information is available for analysis and recovery. The critical difference between self-securing storage and host-controlled versioning (e.g., Elephant [Santry99]) is that intruders can no longer bypass the versioning software by compromising complex OSes or their poorly-protected user accounts. Instead, intruders must compromise single-purpose devices that export only a simple storage interface, and in some configurations, they may have to compromise both.

The first half of this chapter which has been published as “Self-Securing Storage: Protecting Data in Compromised Systems” [Strunk00], describes self-securing storage and our implementation of a self-securing storage server, called S4 as well as the versioning components of self-securing storage. A number of challenges arise when storage devices distrust their clients. Most importantly, it may be difficult to keep all versions of all data for an extended period of time, and it is not acceptable to trust the client to specify what is important to keep. Fortunately, storage densities increase faster than most computer characteristics (100%+ per annum in recent years). Analysis of recent workload studies [Santry99, Vogels99] suggests that it is possible to version all data on modern 30-100GB drives for several weeks. Further, aggressive compression and cross-version differencing techniques can extend the intrusion detection window offered by self-securing storage devices. Other challenges include efficiently encoding the many metadata changes, achieving secure administrative control, and dealing with denial-of-service attacks. Versioning is described in more detail in Section 9.3, as well as in “Metadata Efficiency in a Comprehensive Versioning File System” [Soules03], which was presented at 2nd USENIX Conference on File and Storage Technologies (FAST).

9.2 Self-Securing Storage Systems (S4)

The S4 system addresses the challenges mentioned above with a new storage management structure. Specifically, S4 uses a log-structured object system for data versions and a novel journal-based structure for meta-data versions. In addition to reducing space utilization, journal-based metadata simplifies back-

ground compaction and reorganization for blocks shared across many versions. Experiments with S4 show that the security and data survivability benefits of self-securing storage can be realized with reasonable performance. Specifically, the performance of S4-enhanced NFS is comparable to FreeBSD's NFS for both micro-benchmarks and application benchmarks. The fundamental costs associated with self-securing storage degrade performance by less than 13% relative to similar systems that provide no data protection guarantees.

9.2.1 Intrusion Diagnosis and Recovery

Upon gaining access to a system, an intruder has several avenues of mischief. Most intruders attempt to destroy evidence of their presence by erasing or modifying system log files. Many intruders also install back doors in the system, allowing them to gain access at will in the future. They may also install other software, read and modify sensitive files, or use the system as a platform for launching additional attacks. Depending on the skill with which the intruders hide their presence, there will be some detection latency before the intrusion is discovered by an automated intrusion detection system (IDS) or by a suspicious user or administrator. During this time, the intruders can continue their malicious activities while users continue to use the system, thus entangling legitimate changes with those of the intruders. Once an intrusion has been detected and discontinued, the system administrator is left with two difficult tasks: diagnosis and recovery. Diagnosis is challenging because intruders can usually compromise the \administrator" account on most operating systems, giving them full control over all resources. In particular, this gives them the ability to manipulate everything stored on the system's disks, including audit logs, file modification times, and tamper detection utilities. Recovery is difficult because diagnosis is difficult and because user-convenience is an important issue. This section discusses intrusion diagnosis and recovery in greater detail, and the next section describes how self-securing storage addresses them.

9.2.1.1 Diagnosis

Intrusion diagnosis consists of three phases: detecting the intrusion, discovering what weaknesses were exploited (for future prevention), and determining what the intruder did. All are difficult when the intruder has free reign over storage and the OS. Without the ability to protect storage from compromised operating systems, intrusion detection may be limited to alert users and system administrators noticing odd behavior. Examining the system logs is the most common approach to intrusion detection [Denning87], but when intruders can manipulate the log files, such an approach is not useful. Some intrusion detection systems also look for changes to important system files [Kim94]. Such systems are vulnerable to intruders that can change what the IDS thinks is a "safe" copy. Determining how an intruder compromised the system is often impossible in conventional systems, because he will scrub the system logs. In addition, any exploit tools (utilities for compromising computer systems) that may have been stored on the target machine for use in multi-stage intrusions are usually deleted. The common "solutions" are to try to catch the intruder in the act or to hope that he forgot to delete his exploit tools.

The last step in diagnosing an intrusion is to discover what was accessed and modified by the intruder. This is difficult, because file access and modification times can be changed and system log files can be doctored. In addition, checksum databases are of limited use, since they are effective only for static files.

9.2.1.2 Recovery

Because it is usually not possible to diagnose an intruder's activities, full system recovery generally requires that the compromised machine be wiped clean and reinstalled from scratch. Prior to erasing the entire state of the system, users may insist that data, modified since the intrusion, be saved. The more effort that went into creating the changes, the more motivation there is to keep this data. Unfortunately, as the size and complexity of the data grows, the likelihood that tampering will go unnoticed increases. Foolproof assessment of the modified data is very difficult, and overlooked tampering may hide tainted information or a back door inserted by the intruder.

Upon restoring the OS and any applications on the system, the administrator must identify a backup that was made prior to the intrusion; the most recent backup may not be usable. After restoring data from a pre-intrusion backup, the legitimately modified data can be restored to the system, and users may resume using the system. This process often takes a considerable amount of time—time during which users are denied service.

9.2.2 Self-Securing Storage

Self-securing storage ensures information survival and auditing of all accesses by establishing a security perimeter around the storage device. Conventional storage devices are slaves to host operating systems, relying on them to protect users' data. A self-securing storage device operates as an independent entity, tasked with the responsibility of not only storing data, but also protecting it. This shift of storage security functionality into the storage device's firmware allows data and audit information to be safeguarded in the presence of file server and client system intrusions. Even if the OSes of these systems are compromised and an intruder is able to issue commands directly to the self-securing storage device, the new security perimeter remains intact.

Behind the security perimeter, the storage device ensures data survival by keeping previous versions of the data. This history pool of old data versions, combined with the audit log of accesses, can be used to diagnose and recover from intrusions. This section discusses the benefits of self-securing storage and several core design issues that arise in realizing this type of device.

9.2.2.1 Enabling intrusion survival

Self-securing storage assists in intrusion recovery by allowing the administrator to view audit information and quickly restore modified or deleted files. The audit and version information also helps to diagnose intrusions and detect the propagation of maliciously modified data. Self-securing storage simplifies detection of an intrusion since versioned system logs cannot be imperceptibly altered. In addition, modified system executables are easily noticed. Because of this, self-securing storage makes conventional tamper detection systems obsolete.

Since the administrator has the complete picture of the system's state, from intrusion until discovery, it is considerably easier to establish the method used to gain entry. For instance, the system logs would have normally been doctored, but by examining the versioned copies of the logs, the administrator can see any messages that were generated during the intrusion and later removed. In addition, any exploit tools temporarily stored on the system can be recovered.

Previous versions of system files, from before the intrusion, can be quickly and easily restored by resurrecting them from the history pool. This prevents the need for a complete re-installation of the operating system, and it does not rely on having a recent backup or up-to-date checksums (for tamper detection) of system files. After such restoration, critical data can be incrementally recovered from the history pool. Additionally, by utilizing the storage device's audit log, it is possible to assess which data might have been directly affected by the intruder. The data protection that self-securing storage provides allows easy detection of modifications, selective recovery of tampered files, prevention of data loss due to out-of-date backups, and speedy recovery since data need not be loaded from an off-line archive.

9.2.2.2 Device security perimeter

The device's security model is what makes the ability to keep old versions more than just a user convenience. The security perimeter consists of self-contained software that exports only a simple storage interface to the outside world and verifies each command's integrity before processing it. In contrast, most file servers and client machines run a multitude of services that are susceptible to attack. Since the self-securing storage device is a single-function device, the task of making it secure is much easier; compromising its firmware is analogous to breaking into an IDE or SCSI disk.

The actual protocol used to communicate with the storage device does not affect the data integrity that the new security perimeter provides. The choice of protocol does, however, affect the usefulness of the audit log in terms of the actions it can record and its correctness. For instance, the NFS protocol provides no authentication or integrity guarantees; therefore the audit log may not be able to accurately link a request with its originating client. Nonetheless, the principles of self-securing storage apply equally to "enhanced" disk drives, network-attached storage servers, and file servers.

For network-attached storage devices (as opposed to devices attached directly to a single host system), the new security perimeter becomes more useful if the device can verify each access as coming from both a valid user and a valid client. Such verification allows the device to enforce access control decisions and partially track propagation of tainted data. If clients and users are authenticated, accesses can be tracked to a single client machine, and the device's audit log can yield the scope of direct damage from the intrusion of a given machine or user account.

9.2.2.3 History pool management

The old versions of objects kept by the device comprise the history pool. Every time an object is modified or deleted, the version that existed just prior to the modification becomes part of the history pool. Eventually an old version will age and have its space reclaimed. Because clients cannot be trusted to demarcate versions consisting of multiple modifications, a separate version should be kept for every modification. This is in contrast to versioning file systems that generally create new versions only when a file is closed.

A self-securing storage device guarantees a lower bound on the amount of time that a deprecated object remains in the history pool before it is reclaimed. During this window of time, the old version of the object can be completely restored by requesting that the drive copy forward the old version, thus making a new version. The guaranteed window of time during which an object can be restored is called the detection window. When determining the size of this window, the administrator must examine the trade-off between the detection latency provided by a large window and the extra disk space that is consumed by the proportionally larger history pool.

Although the capacity of disk drives is growing at a remarkable rate, it is still finite, which poses two problems:

1. Providing a reasonable detection window in exceptionally busy systems.
2. Dealing with malicious users that attempt to fill the history pool. (Note that space exhaustion attacks are not unique to self-securing storage. However, device-managed versioning makes conventional user quotas ineffective for limiting them.)

In a busy system, the amount of data written could make providing a reasonable detection window difficult. Fortunately, the analysis in Section 5.2 suggests that multi-week detection windows can be provided in many environments at a reasonable cost. Further, aggressive compression and differencing of old versions can significantly extend the detection window.

Deliberate attempts to overflow the history pool cannot be prevented by simply increasing the space available. As with most denial of service attacks, there is no perfect solution. There are three awed approaches to addressing this type of abuse. The first is to have the device reclaim the space held by the oldest objects when the history pool is full. Unfortunately, this would allow an intruder to destroy information by causing its previous instances to be reclaimed from the over owing history pool. The second flawed approach is to stop versioning objects when the history pool fills; although this will allow recovery of old data, system administrators would no longer be able to diagnose the actions of an intruder or differentiate them from subsequent legitimate changes. The third awed approach is for the drive to deny any action that would require additional versions once the history pool fills; this would result in denial of service to all users (legitimate or not).

Our hybrid approach to this problem is to try to prevent the history pool from being filled by detecting probable abuses and throttling the source machine's accesses. This allows human intervention before the system is forced to choose from the above poor alternatives. Selectively increasing latency and/or decreasing bandwidth allow well-behaved users to continue to use the system even while it is under attack. Experience will show how well this works in practice.

Since the history pool will be used for intrusion diagnosis and recovery, not just recovering from accidental destruction of data, it is difficult to construct a safe algorithm that would save space in the history pool by pruning versions within the detection window. Almost any algorithm that selectively removes versions has the potential to be abused by an intruder to cover his tracks and to successfully destroy/modify information during a break-in.

9.2.2.4 History pool access control

The history pool contains a wealth of information about the system's recent activity. This makes accessing the history pool a sensitive operation, since it allows the resurrection of deleted and overwritten objects. This is a standard problem posed by versioning file systems, but is exacerbated by the inability to selectively delete versions.

There are two basic approaches that can be taken toward access control for the history pool. The first is to allow only a single administrative entity to have the power to view and restore items from the history pool. This could be useful in situations where the old data is considered to be highly sensitive. Having a single tightly controlled key for accessing historical data decreases the likelihood of an intruder gaining access to it. Although this improves security, it prevents users from being able to recover from their own mistakes, thus consuming the administrator's time to restore users' files. The second approach is to allow users to recover their own old objects (in addition to the administrator). This provides the convenience of a user being able to recover their deleted data easily, but also allows an intruder, who obtains valid credentials for a given user, to recover that user's old file versions.

Our compromise is to allow users to selectively make this decision. By choice, a user could thus delete an object, version, or all versions from visibility by anyone other than the administrator, since permanent deletion of data via any other method than aging would be unsafe. This choice allows users to enjoy the benefits of versioning for presentations and source code, while preventing access to visible versions of embarrassing images or unsent e-mail drafts.

9.2.2.5 Administrative access

A method for secure administrative access is needed for the necessary but dangerous commands that a self-securing storage device must support. Such commands include setting the guaranteed detection window, erasing parts of the history pool, and accessing data that users have marked as “unrecoverable.” Such administrative access can be securely granted in a number of ways, including physical access (e.g., flipping a switch on the device) or well-protected cryptographic keys. Administrative access is not necessary for users attempting to recover their own files from accidents. Users’ accesses to the history pool should be handled with the same form of protection used for their normal accesses. This is acceptable for user activity, since all actions permitted for ordinary users can be audited and repaired.

9.2.2.6 Version and administration tools

Since self-securing storage devices store versions of raw data, users and administrators will need assistance in parsing the history pool. Tools for traversing the history must assist by bridging the gap between standard file interfaces and the raw versions that are stored by the device. By being aware of both the versioning system and formats of the data objects, utilities can present interfaces similar to that of Elephant [Santry99], with “time-enhanced” versions of standard utilities such as ls and cp. This is accomplished by extending the read interfaces of the device to include an optional time parameter. When this parameter is specified, the drive returns data from the version of the object that was valid at the requested time. In ad-

dition to providing a simple view of data objects in isolation, intrusion diagnosis tools can utilize the audit log to provide an estimate of damage. For instance, it is possible to see all files and directories that a client modified during the period of time that it was compromised. Further estimates of the propagation of data written by compromised clients are also possible, though imperfect. For example, diagnosis tools may be able to establish a link between objects based on the fact that one was read just before another was written. Such a link between a source file and its corresponding object file would be useful if a user determines that a source file had been tampered with; in this situation, the object file should also be restored or removed. Exploration of such tools will be an important area of future work.

9.2.3 S4 Implementation

S4 is a self-securing storage server that transparently maintains an efficient object-versioning system for its clients. It aims to perform comparably with current systems, while providing the benefits of self-securing storage and minimizing the corresponding space explosion.

RPC Type	Allows Time-Based Access	Description
Create	no	Create an object
Delete	no	Delete an object
Read	yes	Read data from an object
Write	no	Write data to an object
Append	no	Append data to the end of an object
Truncate	no	Truncate an object to a specified length
GetAttr	yes	Get the attributes of an object (S4-specific and opaque)
SetAttr	no	Set the opaque attributes of an object
GetACLByUser	yes	Get an ACL entry for an object given a specific UserID
GetACLByIndex	yes	Get an ACL entry for an object by its index in the object's ACL table
SetACL	no	Set an ACL entry for an object
PCreate	no	Create a partition (associate a name with an ObjectID)
PDelete	no	Delete a partition (remove a name/ObjectID association)
PList	yes	List the partitions
PMount	yes	Retrieve the ObjectID given its name
Sync	not applicable	Sync the entire cache to disk
Flush	not applicable	Removes all versions of all objects between two times
FlushO	not applicable	Removes all versions of an object between two times
SetWindow	not applicable	Adjusts the guaranteed detection window of the S4 device

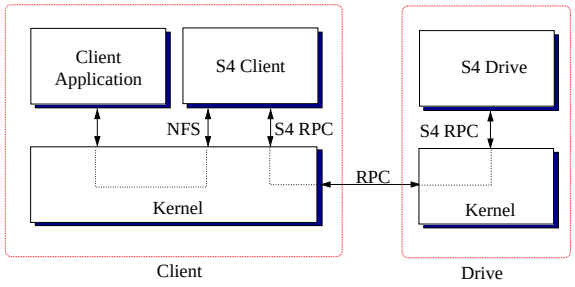
Table 9-1: S4 Remote Procedure Call List - Operations that support time-based access accept a time in addition to the normal parameters; this time is used to find the appropriate version in the history pool. Note that all modifications create new versions without affecting the previous versions.

9.2.3.1 A self-securing object store

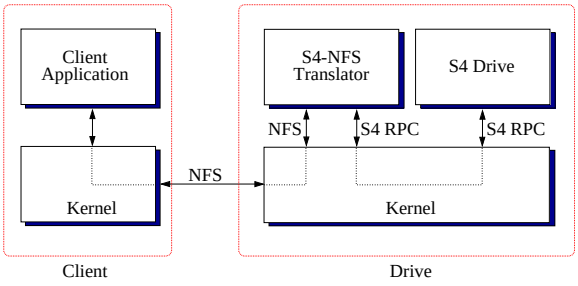
S4 is a network-attached object store with an interface similar to recent object-based disk proposals [Gibson99, OBSD99]. This interface simplifies access control and internal performance enhancement relative to a standard block interface. In S4, objects exist in a flat namespace managed by the “drive” (i.e., the object store). When objects are created, they are given a unique identifier (ObjectID) by the drive, which is used by the client for all future references to that object. Each object has an access control structure that specifies which entities (users and client machines) have permission to access the object. Objects also have metadata, file data, and opaque attribute space (for use by client file systems) associated with them. To enable persistent mount points, an S4drive supports “named objects.” The object names are an association of an arbitrary ASCII string with a particular ObjectID. The table of named objects is implemented as a special S4 object accessed through dedicated partition manipulation RPC calls. This table is versioned in the same manner as other objects on the S4 drive.

S4 RPC interface: Table 9-1 lists the RPC commands supported by the S4 drive. The read-only commands (read, getattr, getacl, plist, and pmount) accept an optional time parameter. When the time is provided, the drive performs the read request on the version of the object that was "most current" at the time specified, provided that the user making the request has sufficient privileges. The ACLs associated with objects have the traditional set of flags, with one addition| the Recovery flag. The Recovery flag determines whether or not a given user may read (recover) an object version from the history pool once it is overwritten or deleted. When this flag is clear, only the device administrator may read this object version once it is pushed into the history pool. The Recovery flag allows users to decide the sensitivity of old versions on a file-by-file basis.

S4/NFS translation: Since one goal of self-securing storage is to provide an enhanced level of security and convenience on existing systems, the prototype minimizes changes to client systems. In keeping with this philosophy, the S4 drive is network-attached and an "S4 client" daemon serves as a user-level file system translator (Figure 9-1a). The S4 client translates requests from a file system on the target OS to S4-specific requests for objects. Because it runs as a user-level process, without operating system modifications, the S4 client should port to different systems easily. The S4 client currently has the ability to translate NFS version 2 requests to S4 requests. The S4 client appears to the local workstation as a NFS server. This emulated NFS server is mounted via the loopback interface to allow only that workstation access to the S4 client. The client receives the NFS requests and translates them into S4 operations. NFSv2 was chosen over version 3 because its client is well supported within Linux, and its lack of write caching allows the drive to maintain a detailed account of client actions.



(a) Baseline S4 (network-attached object store)



(b) S4-enhanced NFS server

Figure 9-1: Two S4 Configurations - This figure shows two S4 configurations that provide self-securing storage via a NFS interface. (a) shows S4 as a network-attached object store with the S4 client daemon translating NFS requests to S4-specific RPCs. (b) shows a self-securing NFS server created by combining the NFS-to-S4 translation and the S4 drive. To support NFSv2 semantics, the client sends an additional RPC to the drive to flush buffered writes to the disk at the end of each NFS operation that modifies the state of one or more objects. Since this RPC does not return until the synchronization is complete, NFSv2 semantics are supported even though the drive normally caches writes.

Figure 9-1 shows two approaches to using the S4 client to serve NFS requests with the S4 drive. The first places the S4 client on the client system, as described previously, and uses the S4 drive as a network-attached storage device. The second incorporates the S4 client functionality into the server, as an NFS-to-S4 translator. This configuration acts as an S4-enhanced NFS server (Figure 9-1b) for normal file system activity, but recovery must still be accomplished through the S4 protocol since the NFS protocol has no notion of "time-based" access.

The implementation of the NFS file system overlays files and directories on top of S4 objects. Objects used as directories contain a list of ASCII filenames and their associated NFS file handles. Objects used as files and symlinks contain the corresponding data. The NFS attribute structure is maintained within the opaque attribute space of each object. When the S4 client receives a NFS request, the NFS file handle (previously constructed by the S4 client) can be directly hashed into the ObjectID of the directory or file. The S4 client can then make requests directly to the drive for the desired data.

Because the client overlays a file system on top of the at object namespace, some file system operations require several drive operations (and hence RPC calls). These sets of operations are analogous to the operations that file systems must perform on block-based devices. To minimize the number of RPC calls necessary, the S4 client aggressively maintains attribute and directory caches (for reads only). The drive also supports batching of setattr, getattr, and sync operations with create, read, write, and append operations.

9.2.3.2 S4 drive internals

The main goals for the S4 drive implementation are to avoid performance overhead and to minimize wasted space, while keeping all versions of all objects for a given period of time. Achieving these goals requires a combination of known and novel techniques for organizing on-disk data.

Log-structuring for efficient writes: Since data within the history pool cannot be over-written, the S4 drive uses a log structure similar to LFS [Rosenblum92]. This structure allows multiple data and metadata updates to be clustered into fewer, larger writes. Importantly, it also obviates any need to move previous versions before writing. In order to prune old versions and reclaim unused segments, S4 includes a background cleaner. While the goal of this cleaner is similar to that of the LFS cleaner, the design must be slightly different. Specifically, deprecated objects cannot be reclaimed unless they have also aged out of the history pool. Therefore, the S4 cleaner searches through the object map for objects with an oldest time greater than the detection window. Once a suitable object is found, the cleaner permanently frees all data and meta-data older than the window. If this clears all of the resources within a segment, the segment can be marked as free and used as a fresh segment for foreground activity.

Journal-based metadata: To efficiently keep all versions of object metadata, S4 uses journal-based metadata, which replaces most instances of metadata with compact journal entries. Because clients are not trusted to notify S4 when objects are closed, every update creates a new version and thus new metadata. For example, when data pointed to by indirect blocks is modified, the indirect blocks must be versioned as well. In a conventional versioning system, a single update to a triple-indirect block could require four new blocks as well as a new inode. Early experiments with this type of versioning system showed that modifying a large file could cause up to a 4x growth in disk usage. Conventional versioning file systems avoid this performance problem by only creating new versions when a file is closed. In order to significantly reduce these problems, S4 encodes metadata changes in a journal that is maintained for the duration of the detection window. By persistently keeping journal entries of all metadata changes, metadata writes can be safely delayed and coalesced, since individual inode and indirect block versions can be recreated from the journal. To avoid rebuilding an object's current state from the journal during normal operation, an object's metadata is check pointed to a log segment before being evicted from the cache. Unlike conventional journaling, such check pointing does not prune journal space; only aging may prune space.

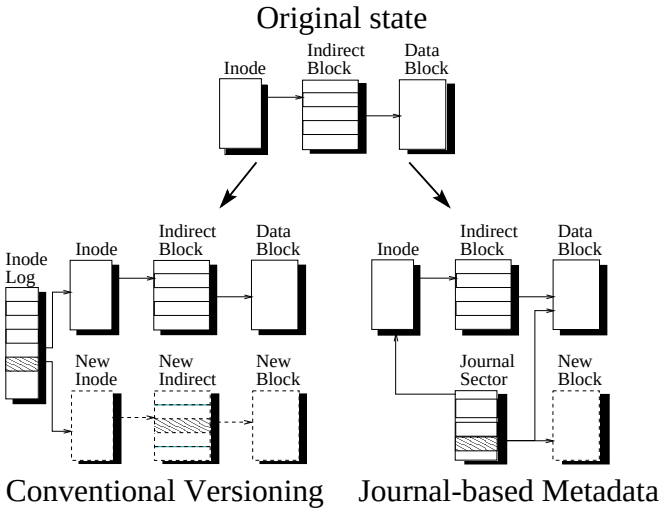


Figure 9-2: Efficiency of Metadata Versioning - The above figure compares metadata management in a conventional versioning system to S4's journal-based metadata approach. When writing to an indirect block, a conventional versioning system allocates a new data block, a new indirect block, and a new inode. Also, the identity of the new inode must be recorded (e.g., in an Elephant-like inode log). With journal-based metadata, a single journal entry suffices, pointing to both the new and old data blocks.

Figure 9-2 depicts the difference in disk

space usage between journal-based metadata and conventional versioning when writing data to an indirect data block. In addition to the entries needed to describe metadata changes, a checkpoint entry is needed. This checkpoint entry denotes writing a consistent copy of all of an object's metadata to disk. It is necessary to have at least one checkpoint of an object's metadata on disk at all times, since this is the starting point for all time-based and crash recovery recreations.

Storing an object's changes within the log is done using journal sectors. Each journal sector contains the packed journal entries that refer to a single object's changes made within that segment. The sectors are identified by segment summary information. Journal sectors are chained together backward in time to allow version reconstruction.

Journal-based metadata can also simplify cross-version differential compression [Burns96]. Since the blocks changed between versions are noted within each entry, it is easy to find the blocks that should be compared. Once the differencing is complete, the old blocks can be discarded, and the difference left in its place. For subsequent reads of old versions, the data for each block must be recreated as the entries are traversed. Cross-version differencing of old data will often be effective in reducing the amount of space used by old versions. Adding differencing technology into the S4 cleaner is an area of future work.

Audit log: In addition to maintaining previous object versions, S4 maintains an append-only audit log of all requests. This log is implemented as a reserved object within the drive that cannot be modified except by the drive itself. However, it can be read via RPC operations. The data written to the audit log includes command arguments as well as the originating client and user. All RPC operations (read, write, and administrative) are logged. Since the audit log may only be written by the drive front end, it need not be versioned, thus increasing space efficiency and decreasing performance costs.

9.2.4 Evaluation of self-securing storage

This section evaluates the feasibility of self-securing storage. Experiments with S4 indicate that comprehensive versioning and auditing can be performed without a significant performance impact. Also, estimates of capacity growth, based on reported workload characterizations, indicate that history windows of several weeks can easily be supported in several real environments.

9.2.4.1 Performance

The main performance goal for S4 is to be comparable to other networked file systems while offering enhanced security features. This section demonstrates that this goal is achieved and also explores the overheads specifically associated with self-securing storage features.

Experimental Setup: The four systems used in the experiments had the following configurations: (1) a S4 drive running on RedHat 6.1 Linux communicating with a Linux client over S4 RPC through the S4 client module (Figure 9-1a), (2) a S4-enhanced NFS server running on RedHat 6.1 Linux communicating with a Linux client over NFS (Figure 9-1b), (3) a FreeBSD 4.0 server communicating with a Linux client over NFS, and (4) a RedHat 6.1 Linux server communicating with a Linux client over NFS. Since Linux NFS does not comply with the NFSv2 semantics of committing data to stable storage before operation completion, the Linux server's file system was mounted synchronously to approximate NFS semantics. In all cases, NFS was configured to use 4KB read/write transfer sizes, the only option supported by Linux. The FreeBSD NFS configuration exports a BSD FFS file system, while the Linux NFS configuration exports an ext2 file system. All experiments were run five times and have a standard deviation of less than 3% of the mean. The S4 drives were configured with a 128MB buffer cache and a 32MB object cache. The Linux and FreeBSD NFS servers' caches could grow to fill local memory (512MB). In all experiments, the client system has a 550MHz Pentium III, 128MB RAM, and a 3Com 3C905B 100Mb network adapter. The servers have a 600MHz Pentium III, 512MB RAM, a 9GB 10; 000RPM Ultra2 SCSI Seagate Cheetah drive, an Adaptec AIC-7896/7 Ultra2 SCSI controller, and an Intel Ether-Express Pro100 100Mb network adapter. The client and server are on the same subnet and are connected by a 100Mb

network switch. All versions of Linux use an unmodified 2.2.14 kernel, and the BSD system uses a stock FreeBSD 4.0 installation.

To evaluate performance for common workloads, results from two application benchmarks are presented: the PostMark benchmark [Katcher97] and the SSH-build benchmark [Ylonen96]. These benchmarks crudely represent Internet server and software development workloads, respectively.

PostMark was designed to measure the performance of a file system used for electronic mail, netnews, and web based services. It creates a large number of small randomly-sized files (between 512B and 9KB) and performs a specified number of transactions on them. Each transaction consists of two sub-transactions, with one being a create or delete and the other being a read or append. The default configuration used for the experiments consists of 20,000 transactions on 5,000 files, and the biases for transaction type are equal.

The SSH-build benchmark was constructed as a replacement for the Andrew file system benchmark [Howard88]. It consists of 3 phases: The unpack phase, which unpacks the compressed tar archive of SSH v1.2.27 (approximately 1MB in size before decompression), stresses metadata operations on files of varying sizes. The configure phase consists of the automatic generation of header files and Makefiles, which involves building various small programs that check the existing system configuration. The build phase compiles, links, and removes temporary files. This last phase is the most CPU intensive, but it also generates a large number of object files and a few executables.

Comparison of the Servers: To gauge the overall performance of S4, the four systems described earlier were compared. As hoped, S4 performs comparably to the existing NFS servers. Figure 9-3 shows the results of the PostMark benchmark. The times for both the creation (time to create the initial 5000 files) and transaction phases of PostMark are shown for each system. The S4 systems' performance is similar to both BSD and Linux NFS performance, doing slightly better due to their log structured layout.

The times of SSH-build's three phases are shown in Figure 9-4. Performance is similar across the S4 and BSD configurations. The superior performance of the Linux NFS server in the configure stage is due to a much lower number of write I/Os than in the BSD and S4 servers, apparently due to a flaw in the synchronous mount option under Linux.

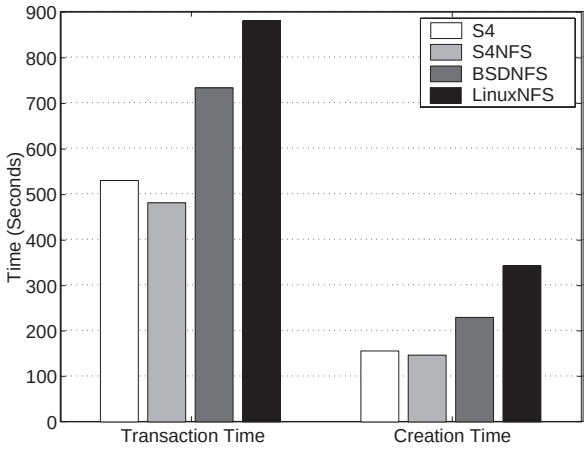


Figure 9-3: PostMark Benchmark Unpack Time (seconds).

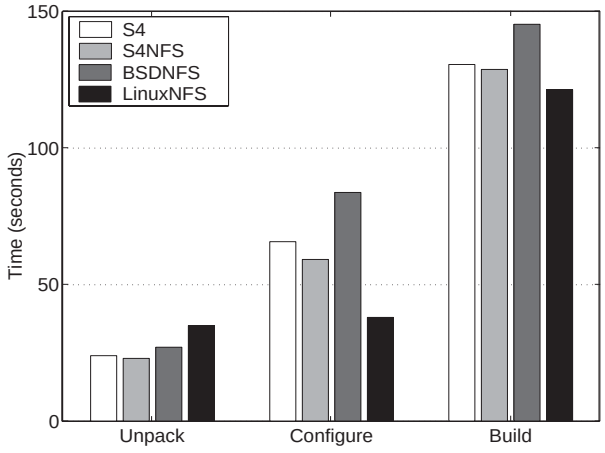


Figure 9-4: SSH-build Benchmark

Overhead of the S4 Cleaner: In addition to the more visible process of creating new versions, S4 must eventually garbage collect data that has expired from the history pool. This garbage collection comes at a cost. The potential overhead of the cleaner was measured by running the PostMark benchmark with

50,000 transactions on increasingly large sets of initial files. For each set of initial files, the benchmark was run once with the cleaner disabled and once with the cleaner competing with foreground activity.

The results shown in Figure 9-5 represent PostMark running with the initial set of files filling between 2% and 90% of a 2GB disk. As expected, when the working set increases, performance of the normal S4 system degrades due to increasingly poor cache and disk locality. The sharp drop in the graph from 2% to 10% is caused by the fact that the set of files and data expands beyond the bounds of the drive's cache.

Although the S4 cleaner is slightly different, it was expected to behave similarly to a standard LFS cleaner, which has up to an approximate 34% decrease in performance [Seltzer95]. The S4 cleaner is slightly more intrusive, degrading performance by approximately 50% in the worst case. The greater degradation is attributed mainly to the additional reads necessary when cleaning objects rather than segments. In addition, the S4 cleaner has not been tuned and does not include known techniques for reducing cleaner performance problems [Matthews97].

Overhead of the S4 audit log: In addition to versioning, self-securing storage devices keep an audit log of all connections and commands sent to the drive. Recording this audit log of events has some cost. In the worst case, all data written to the disk belongs to the audit log. In this case, one disk write is expected approximately every 750 operations. In the best case, large writes, the audit log overhead is almost non-existent, since the writes of the audit log blocks are hidden in the segment writes of the requests. For the macro-benchmarks, the performance penalty ranged between 1% and 3%.

For a more focused view of this overhead, a set of micro-benchmarks was run with audit logging enabled and disabled. The micro-benchmarks proceed in three phases: creation of 10,000 1KB files (split across 10 directories), reads of the newly created files in creation order, and deletion of the files in creation order.

Figure 9-6 shows the results. The create and delete phases exhibit a 2.8% and 2.9% decrease in performance, respectively, and the read phase exhibits a 7.2% decrease in performance. Read performance suffers a larger penalty because the audit log blocks become interwoven with the data blocks in the create phase. This reduces the number of files packed into each segment, which in turn increases the number of segment reads required.

Fundamental Performance Costs: There are three fundamental performance costs of self-securing storage: versioning, auditing, and garbage collection. Versioning can be achieved at virtually no cost by combining journal-based metadata with the LFS structure. Auditing creates a small performance penalty of

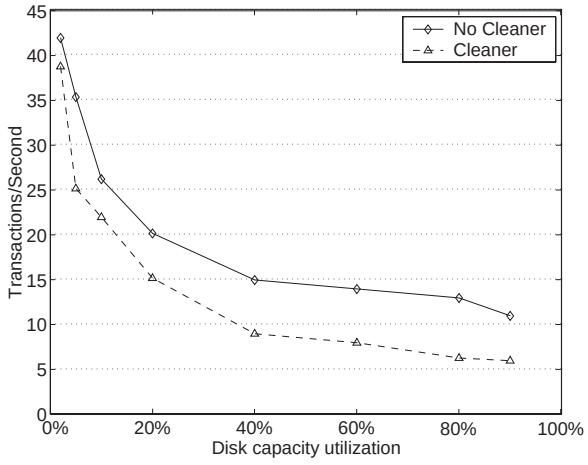


Figure 9-5: Overhead of foreground cleaning in S4. This figure shows the transaction performance of S4 running the PostMark benchmark with varying capacity utilizations. The solid line shows system performance on a system without cleaning. The dashed line shows system performance in the presence of continuous foreground cleaner activity.

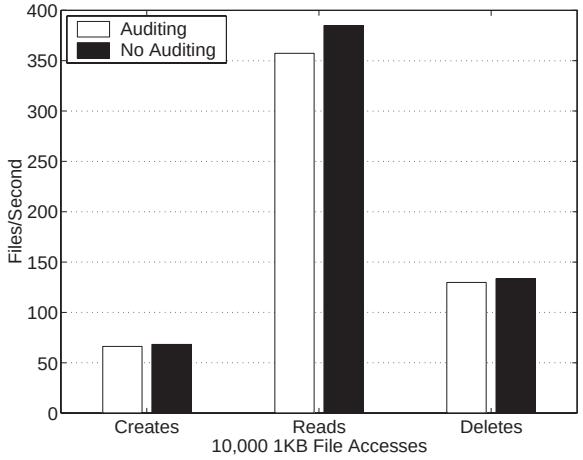


Figure 9-6: Auditing Overhead in S4. This figure shows the impact on small file performance caused by auditing incoming client requests.

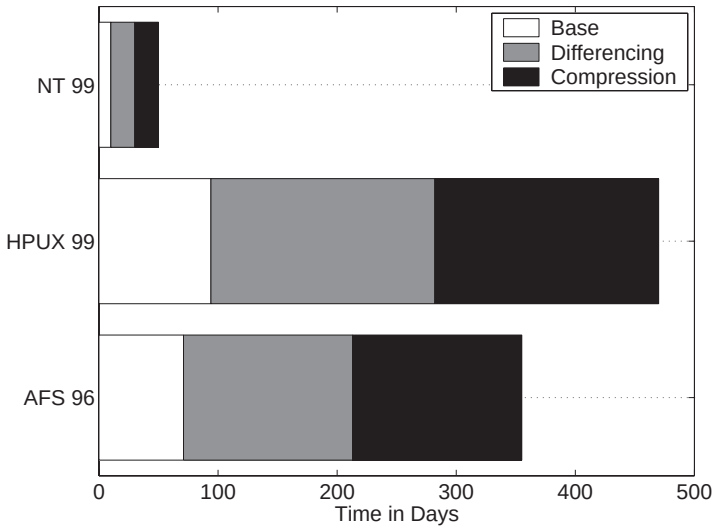


Figure 9-7: Projected Detection Window -The expected detection window that could be provided by utilizing 10GB of a modern disk drive. This conservative history pool would consume only 20% of a 50GB disk's total capacity. The baseline number represents the projected number of days worth of history information that can be maintained within this 10GB of space. The gray regions show the projected increase that cross-version differencing would provide. The black regions show the further increase expected from using compression in addition to differencing.

1% to 3%, according to application benchmarks. The final performance cost, garbage collection, is more difficult to quantify. The extra overhead of S4 cleaning in comparison to standard LFS cleaning comes mainly from the difference in utilized space due to the history pool. The worst-case performance penalty for garbage collection in S4 can be estimated by comparing the cleaning overhead at two space utilizations: the space utilized by the active set of objects and the space utilized by the active set combined with the history pool. For example, assume that the active set utilizes 60% of the drive's space and the history pool another 20%. For PostMark, the cleaning overhead is the difference between cleaning performance and standard performance seen at a given space utilization in Figure 9-5. For 60% utilization, the cleaning overhead is 43%. For 80% utilization, it is 53%. Thus, in this example, the extra cleaning overhead caused by keeping the history pool is 10%.

There are several possibilities for reducing cleaner overhead for all space utilizations. With expected detection windows ranging into the hundreds of days, it is likely that the history pool can be extended until such a time that the drive becomes idle. During idle time, the cleaner can run with no observable overhead [Blackwell95]. Also, recent research into technologies such as freeblock scheduling offers standard LFS cleaning at almost no cost [Lumb00]. This technique could be extended for cleaning in S4.

9.2.4.2 Capacity Requirements

To evaluate the size of the detection window that can be provided, three recent workload studies were examined. Figure 9-7 shows the results of approximations based on worst-case write behavior. Spasojevic and Satyanarayanan's AFS trace study [Spasojevic96] reports approximately 143MB per day of write traffic per file server. The AFS study was conducted using 70 servers (consisting of 32; 000 cells) distributed across the wide area, containing a total of 200GB of data. Based on this study, using just 20% of a modern 50GB disk would yield over 70 days of history data. Even if the writes consume 1GB per day per server, as was seen by Vogels' Windows NT file usage study [Vogels99], 10 days worth of history data can be provided. The NT study consisted of 45 machines split into personal, shared, and administrative domains running workloads of scientific processing, development, and other administrative tasks. Santry, et al. [Santry99] report a write data rate of 110MB per day. In this case, over 90 days of data could be kept. Their environment consisted of a single file system holding 15GB of data that was being used by a dozen researchers for development.

Much work has been done in evaluating the efficiency of differencing and compression [Burns96, Burrows94, Burrows92]. To briefly explore the potential benefits for S4, its code base was retrieved from the CVS repository at a single point each day for a week. After compiling the code, both differencing and differencing with compression were applied between each tree and its direct neighbor in time using Xdelta [MacDonald00, MacDonald98]. After applying differencing, the space efficiency increased by

200%. Applying compression added an additional 200% for a total space efficiency of 500%. These results are in line with previous work. Applying these estimates to the above workloads indicates that a 10GB history pool can provide a detection window of between 50 and 470 days.

9.2.5 Discussion

This section discusses several important implications of self-securing storage.

Selective versioning: There are data that users would prefer not to have backed up at all. The common approach to this is to store them in directories known to be skipped by the backup system. Since one of the goals of S4 is to allow recovery of exploit tools, it does not support designating objects as non-versioned. A system may be configured with non-S4 partitions to support selective versioning. While this would provide a way to prevent versioning of temporary files and other non-critical data, it would also create a location where an intruder could temporarily store exploit tools without fear that they will be recovered.

Versioning vs. snapshots: Self-securing storage can be implemented with frequent copy-on-write snapshots [Hitz94, Howard88, Lee96] instead of versioning, so long as snapshots are kept for the full detection window. Although the audit log can still provide a record of what blocks are changed, snapshots often will not allow administrators to recover short-lived files (e.g., exploit tools) or intermediate versions (e.g., system log file updates). Also, legitimate changes are only guaranteed to survive malicious activity if they survive to the next snapshot time. Of course, the potential scope of such problems can be reduced by shrinking the time between snapshots. The comprehensive versioning promoted in this paper represents the natural end-point of such shrinking—every modification creates a new snapshot.

Versioning file systems vs. self-securing storage: Versioning file systems excel at providing users with a safety net for recovery from accidents. They maintain old file versions long after they would be reclaimed by the S4 system, but they provide little additional system security. This is because they rely on the host's OS for security and aggressively prune apparently insignificant versions. By combining self-securing storage with long-term landmark versioning [Santry92], recovery from users' accidents could be enhanced while also maintaining the benefits of intrusion survival.

Self-securing storage for databases: Most databases log all changes in order to protect internal consistency in the face of system crashes. Some institutions also retain these logs for long-term auditing purposes. All information needed to understand and recover from malicious behavior can be kept, in database-specific form, in these logs. Self-securing storage can increase the post-intrusion recoverability of database systems in two ways: (1) by preventing undetectable tampering with stored log records, and (2) by preventing undetectable changes to data that bypass the log. After an intrusion, self-securing storage allows a database system to verify its log's integrity and confirm that all changes are correctly rejected in the log—the database system can then safely use its log for subsequent recovery.

Client-side cache effects: In order to improve efficiency, most client systems use caches to minimize storage latencies. This is at odds with the desire to have storage devices audit users' accesses and capture exploit tools. Client-side read caches hide data dependency information that would otherwise be available to the drive in the form of reads followed quickly by writes. However, this information could be provided by client systems as (questionable) hints during writes. Write caches cause a more serious problem when files are created then quickly deleted, thus never being sent to the drive. This could cause difficulties with capturing exploit tools, since they may never be written to the drive. Although client cache effects may obscure some of the activity in the client system, data that are stored on a self-securing storage device are still completely protected.

Object-based vs. block-based storage: Implementing a self-securing storage device with a block interface adds several difficulties. Since objects are designed to contain one data item (file or directory), enforcing access control at this level is much more manageable than attempting to assign permissions on a per-block

basis. In addition, maintaining versions of objects as a whole, rather than having to collect and correlate individual blocks, simplifies recovery tools and internal reorganization mechanisms.

Multi-device coordination: Multi-device coordination is necessary for operations such as striping data or implementing RAID across multiple self-securing disks or file servers. In addition to the coordination necessary to ensure that multiple copies of data are synchronized, recovery operations must also coordinate old versions. On the other hand, clusters of self-securing storage devices could maintain a single history pool and balance the load of versioning objects. Note that a self-securing storage device containing several disks (e.g., a self-securing disk array) does not have these issues. Additionally, it has the ability to keep old versions and current data on separate disks.

9.3 Metadata Efficiency in Versioning File Systems

Self-securing storage [Strunk00] uses versioning to enable storage servers to internally retain file versions and provide detailed information for post-intrusion diagnosis and recovery of compromised client systems [Strunk02]. We envision self-securing storage servers that retain every version of every file, where every modification (e.g., a WRITE operation or an attribute change) creates a new version. Such *comprehensive versioning* maximizes the information available for post-intrusion diagnosis. Specifically, it avoids pruning away file versions, since this might obscure intruder actions. For self-securing storage, *pruning techniques* are particularly dangerous when they rely on client-provided information, such as CLOSE operations—the versioning is being done specifically to protect stored data from malicious clients.

Obviously, finite storage capacities will limit the duration of time over which comprehensive versioning is possible. To be effective for intrusion diagnosis and recovery, this duration must be greater than the intrusion detection latency (i.e., the time from an intrusion to when it is detected). We refer to the desired duration as the *detection window*. In practice, the duration is limited by the rate of data change and the space efficiency of the versioning system. The rate of data change is an inherent aspect of a given environment, and an analysis of several real environments suggests that detection windows of several weeks or more can be achieved with only a 20% cost in storage capacity [Strunk00].

Section 9.2 describes a prototype self-securing storage system. By using standard copy-on-write and a log-structured data organization, the prototype provided comprehensive versioning with minimal performance overhead (<10%) and reasonable space efficiency. In that work, we discovered that a key design requirement is efficient encoding of metadata versions (the additional information required to track the data versions). While copy-on-write reduces data versioning costs, conventional versioning implementations still involve one or more new metadata blocks per version. On average, the metadata versions require as much space as the versioned data, halving the achievable detection window. Even with less comprehensive versioning, such as Elephant [Santry99] or VMS [McCoy90], the metadata history can become almost (≈80%) as large as the data history.

This half of the chapter describes and evaluates two methods of storing metadata versions more compactly: journal-based metadata and multiversion b-trees. Journal-based meta-data encodes each version of a file's metadata in a journal entry. Each entry describes the difference between two versions, allowing the system to roll-back to the earlier version of the metadata. Multiversion b-trees retain all versions of a metadata structure within a single tree. Each entry in the tree is marked with a timestamp indicating the time over which the entry is valid.

The two mechanisms have different strengths and weaknesses. We discuss these and describe how both techniques are integrated into a comprehensive versioning file system called CVFS. CVFS uses journal-based meta-data for inodes and indirect blocks to encode changes to attributes and file data pointers; doing so reduces the space used for their histories by 80%. CVFS implements directories as multiversion b-trees to encode additions and removals of directory entries; doing so reduces the space used for their histories by 99%. Combined, these mechanisms nearly double the potential detection window over conventional versioning mechanisms, without increasing the access time to current versions of the data.

Journal-based metadata and multiversion b-trees are also valuable for conventional versioning systems. Using these mechanisms with on-close versioning and snapshots would provide similar reductions in versioned metadata. For on-close versioning, this reduces the total space required by nearly 35%, thereby reducing the pressure to prune version histories. Identifying solid heuristics for such pruning remains an open area of research [Santry99], and less pruning means fewer opportunities to mistakenly prune important versions.

9.3.1 Versioning and Space Efficiency

Every modification to a file inherently results in a new version of the file. Instead of replacing the previous version with the new one, a *versioning file system* retains both. Users of such a system can then access any historical versions that the system keeps as well as the most recent one. This section discusses uses of versioning, techniques for managing the associated capacity costs, and our goal of minimizing the metadata required to track file versions.

9.3.1.1 Uses of Versioning

File versioning offers several benefits to both users and system administrators. These benefits can be grouped into three categories: recovery from user mistakes, recovery from system corruption, and analysis of historical changes. Each category stresses different features of the versioning system beneath it.

Recovery from user mistakes: Human users make mistakes, such as deleting or erroneously modifying files. Versioning can help [Hagmann87, McCoy90, Santry99]. Recovery from such mistakes usually starts with some a priori knowledge about the nature of the mistake. Often, the exact file that should be recovered is known. Additionally, there are only certain versions that are of any value to the user; intermediate versions that contain incomplete data are useless. Therefore, versioning aimed at recovery from user mistakes should focus on retaining key versions of important files.

Recovery from system corruption: When a system becomes corrupted, administrators generally have no knowledge about the scope of the damage. Because of this, they restore the entire state of the file system from some well-known "good" time. A common versioning technique to help with this is the online *snapshot*. Like a backup, a snapshot contains a version of every file in the system at a particular time. Thus, snapshot systems present sets of known-valid system images at a set of well-known times.

Analysis of historical changes: A history of versions can help answer questions about how a file reached a certain state. For example, version control systems (e.g., RCS [Tichy80], CVS [Grune]) keep a complete record of committed changes to specific files. In addition to selective recovery, this record allows developers to figure out who made specific changes and when those changes were made. Similarly, self-securing storage seeks to enable post-intrusion diagnosis by providing a record of what happened to stored files before, during, and after an intrusion. We believe that every version of every file should be kept. Otherwise, intruders who learn the pruning heuristic will leverage this information to prune any file versions that might disclose their activities. For example, intruders may make changes and then quickly revert them once damage is caused in order to hide their tracks. With a complete history, administrators can determine which files were changed and estimate damage. Further, they can answer (or at least construct informed hypotheses for) questions such as "When and how did the intruder get in?" and "What was their goal?" [Strunk02].

9.3.1.2 Pruning Heuristics

A true comprehensive versioning system keeps all versions of all files for all time. Such a system could support all three goals described above. Unfortunately, storing this much information is not practical. As a result, all versioning systems use *pruning heuristics*. These pruning heuristics determine when versions should be created and when they should be removed. In other words, pruning heuristics determine which versions to keep from the total set of versions that would be available in a comprehensive versioning system.

Common Heuristics: A common pruning technique in versioning file systems is *on-close* versioning. This technique keeps only the last version of a file from each session; that is, each CLOSE of a file creates a distinct version. For example, the VMS file system [McCoy90] retains a fixed number of versions for each file. VMS's pruning heuristic creates a version after each CLOSE of a file and, if the file already has the maximum number of versions, removes the oldest remaining version of the file. The more recent Elephant file system [Santry99] also creates new versions after each CLOSE; however, it makes additional pruning decisions based on a set of rules derived from observed user behavior.

Version control systems prune in two ways. First, they retain only those versions explicitly committed by a user. Second, they retain versions for only an explicitly-chosen subset of the files on a system.

By design, snapshot systems like WAFL [Hitz94] and Venti/Plan9 [Quinlan02] prune all of the versions of files that are made between snapshots. Generally, these systems only create and delete snapshots on request, meaning that the system's administrator decides most aspects of the pruning heuristic.

Information Loss: Pruning heuristics act as a form of lossy compression. Rather than storing every version of a file, these heuristics throw some data away to save space. The result is that, just as a JPEG file loses some of its visual clarity with lossy compression, pruning heuristics reduce the clarity of the actions that were performed on the file.

Although this loss of information could result in annoyances for users and administrators attempting to recover data, the real problem arises when versioning is used to analyze historical changes. When versioning for intrusion survival, as in the case of self-securing storage, pruning heuristics create holes in the administrator's view of the system. Even creating a version on every CLOSE is not enough, as malicious users can leverage this heuristic to hide their actions (e.g., storing exploit tools in an open file and then truncating the file to zero before closing it).

To avoid traditional pruning heuristics, self-securing storage employs comprehensive versioning over a fixed window of time, expiring versions once they become older than the given window. This detection window can be thought of as the amount of time that an administrator has to detect, diagnose, and recover from an intrusion. As long as an intrusion is detected within the window, the administrator has access to the entire sequence of modifications since the intrusion.

9.3.1.3 Lossless Version Compression

For a system to avoid pruning heuristics, even over a fixed window of time, it needs some form of lossless version compression. Lossless version compression can also be combined with pruning heuristics to provide further space reductions in conventional systems. To maximize the benefits, a system must attempt to compress both versioned data and versioned metadata.

Data: Data block sharing is a common form of loss-less compression in versioning systems. Unchanged data blocks are shared between versions by having their individual metadata point to the same physical block. Copy-on-write is used to avoid corrupting old versions if the block is modified.

An improvement on block sharing is byte-range differencing between versions. Rather than keeping the data blocks that have changed, the system keeps the bytes that have changed [MacDonald98]. This is especially useful in situations where a small change is made to the file. For example, if a single byte is inserted at the beginning of a file, a block sharing system keeps two full copies of the entire file (since the data of every block in the file is shifted forward by one byte); for the same scenario, a differencing system only stores the single byte that was added and a small description of the change.

Another recent improvement in data compression is hash-based data storage [Muthitacharoen01, Quinlan02]. These methods recognize identical blocks or ranges of data across the system and store only one copy of the data. This method is quite effective for snapshot versioning systems, and could likely be applied to other versioning systems with similar results.

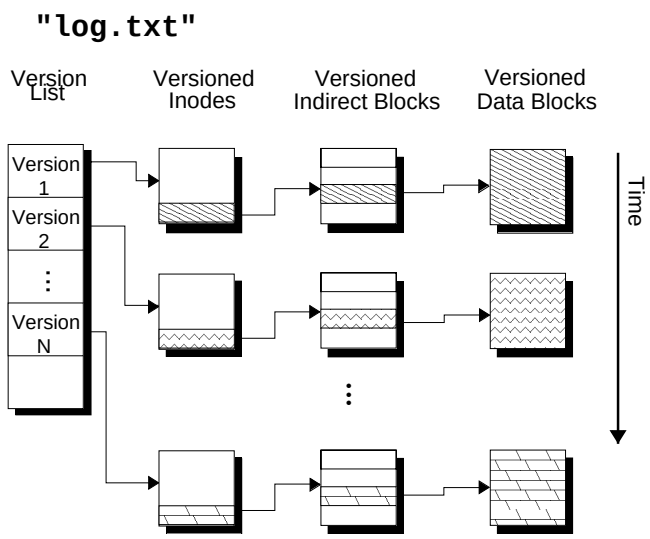


Figure 9-8: Conventional versioning system. In this example, a single logical block of file "log.txt" is overwritten several times. With each new version of the data block, new versions of the indirect block and inode that reference it are created. Notice that although only a single pointer has changed in both the indirect block and the inode, they must be rewritten entirely, since they require new versions. The system tracks each version with a pointer to that version's inode.

Metadata: Conventional versioning file systems keep a full copy of the file metadata with each version. While it simplifies version access, this method quickly exhausts capacity, since even small changes to file data or attributes result in a new copy of the metadata.

Figure 9-8 shows an example of how the space overhead of versioned metadata can become a problem in a conventional versioning system. In this example, a program is writing small log entries to the end of a large file. Since several log entries fit within a single data block, appending entries to the end of the file produces several different versions of the same block. Because each versioned data block has a different location on disk, the system must create a new version of the indirect block to track its location. In addition, the system must write a new version of the inode to track the location of the versioned indirect block. Since any data or metadata change will always result in a new version of the inode, each version is tracked using a pointer to that version's inode. Thus, writing a single data block results in a new indirect block, a new inode, and an entry in the version list, resulting in more metadata being written than data.

Access patterns that create such metadata versioning problems are common. Many applications create or modify files piece by piece. In addition, distributed file systems such as NFS create this behavior by breaking large updates of a file into separate, block-sized updates. Since there is no way for the server to determine if these block-sized writes are one large update or several small ones, each must be treated as a separate update, resulting in several new versions of the file.

Again, the solution to this problem is some form of differencing between the versions. Mechanisms for creating and storing differences of metadata versions are the main focus of this work.

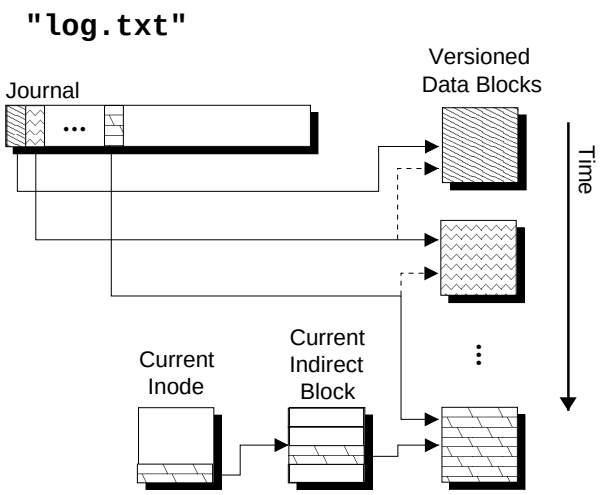
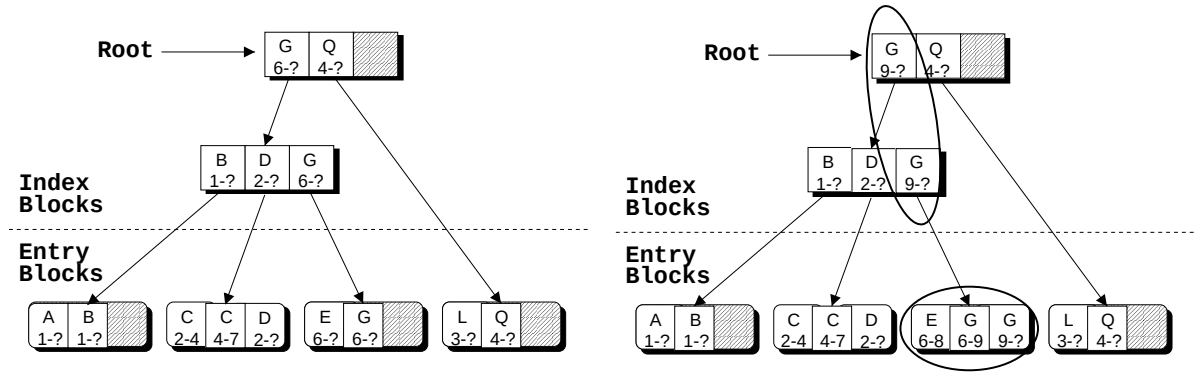


Figure 9-9: Journal-based metadata system. Just as in Figure 9-8, this figure shows a single logical block of file "log.txt" being overwritten several times. Journal-based metadata retains all versions of the data block by recording each in a journal entry. Each entry points to both the new block and the block that was overwritten. Only the current version of the inode and indirect block are kept, significantly reducing the amount of space required for metadata.

9.3.1.4 Objective

In a perfect world we could keep all versions of all files for an infinite amount of time with no impact on performance. This is obviously not possible. The objective of this work is to minimize the space overhead of versioned metadata. For self-securing storage, doing so will increase the detection window. For other versioning purposes, doing so will



(a) initial tree structure

(b) After removal of E and update of G.

Figure 9-10: Multiversion b-tree. This figure shows the layout of a multiversion b-tree. Each entry of the tree is designated by a _user-key, timestamp_ tuple which acts as a key for the entry. A question mark (?) in the timestamp indicates that the entry is valid through the current time. Different versions of an entry are separate entries using the same user-key with different timestamps. Entries are packed into entry blocks, which are tracked using index blocks. Each index pointer holds the key of the last entry along the subtree that it points to.

reduce the pressure to prune. Because this space reduction will require compressing metadata versions, it is also important that the performance overhead of both version creation and version access be minimized.

9.3.2 Efficient Metadata Versioning

One characteristic of versioned metadata is that the actual changes to the metadata between versions are generally quite small. In Figure 9-8, although an inode and an indirect block are written with each new version of the file, the only change to the metadata is an update to a single block pointer. The system can leverage these small changes to provide much more space-efficient metadata versioning. This section describes two methods that leverage small metadata modifications, and Section 9.3.3 describes an implementation of these solutions.

9.3.2.1 Journal-based Metadata

Journal-based metadata maintains a full copy of the current version's metadata and a journal of each previous metadata change. To recreate old versions of the meta-data, each change is undone backward through the journal until the desired version is recreated. This process of undoing metadata changes is referred to as *journal rollback*.

Figure 9-9 illustrates how journal-based metadata works in the example of writing log entries. Just as in Figure 9-8, the system writes a new data block for each version; however, in journal-based metadata, these blocks are tracked using small journal entries that track the locations of the new and old blocks. By keeping the current version of the metadata up-to-date, the journal entries can be rolled-back to any previous version of the file.

In addition to storing version information, the journal can be used as a write-ahead log for metadata consistency, just as in a conventional journaling file system. To do so, the new block pointer must be recorded in addition to the old. Using this, a journal-based metadata implementation can safely maintain the current version of the metadata in memory, flushing it to disk only when it is forced from the cache.

Space vs. Performance: Journal-based metadata is more space efficient than conventional versioning. However, it must pay a performance penalty for recreating old versions of the meta-data. Since each entry written between the current version and the requested version must be read and rolled-back, there is a linear relation between the number of changes to a file and the performance penalty for recreating old versions.

One way the system can reduce this overhead is to *checkpoint* a full copy of a file's metadata to the disk occasionally. By storing checkpoints and remembering their locations, a system can start journal roll-back from the closest checkpoint in time rather than always starting with the current version. The frequency with which these checkpoints are written dictates the space/performance trade-off. If the system keeps a checkpoint with each modification, journal-based metadata performs like a conventional versioning scheme (using the most space, but offering the best back-in-time performance). However, if no checkpoints are written, the only full instance of the metadata is the current version, resulting in the lowest space utilization but reduced back-in-time performance.

9.3.2.2 Multiversion B-trees

A multiversion b-tree is a variation on standard b-trees that keeps old versions of entries in the tree [Becker96]. As in a standard b-tree, an entry in a multiversion b-tree contains a key/data pair; however, the key consists of both a user-defined key and the time at which the entry was written. With the addition of this time-stamp, the key for each version of an entry becomes unique. Having unique keys means that entries within the tree are never overwritten; therefore, multiversion b-trees can have the same basic structure and operations as a standard b-tree. To facilitate current version lookups, entries are sorted first by the user-defined key and then by the timestamp.

Figure 9-10a shows an example of a multiversion b-tree. Each entry contains both the user-defined key and the time over which the entry is valid. The entries are packed into entry blocks, which act as the leaf nodes of the tree. The entry blocks are tracked using index blocks, just as in standard b+trees. In this example, each pointer in the index block references the last entry of the subtree beneath it. So in the case of the root block, the *G* subtree holds all entries with values less than or equal to *G*, with ⟨*G*, 6–?⟩ as its last entry. The *Q* subtree holds all entries with values between *G* and *Q*, with ⟨*Q*, 4–?⟩ as its last entry.

Figure 9-10b shows the tree after a remove of entry *E* and an update to entry *G*. When entry *E* is removed at time *8*, the only change is an update to the entry's timestamp. This indicates that *E* is only valid from time *6* through time *8.* When entry *G* is updated at time *9*, a new entry is created and associated with the new data. Also, the old entry for *G* must be updated to indicate its bounded window of validity. In this case, the index blocks must also be updated to reflect the new state of the subtree, since the last entry of the subtree has changed.

Since both current and history entries are stored in the same tree, accesses to old and current versions have the same performance. For this reason, large numbers of history entries can decrease the performance of accessing current entries.

9.3.2.3 Solution Comparison

Both journal-based metadata and multiversion b-trees reduce the space utilization of versioning but incur some performance penalty. Journal-based metadata pays with reduced back-in-time performance. Multiversion b-trees pay with reduced current version performance.

Because the two mechanisms have different drawbacks, they each perform certain operations more efficiently. As mentioned above, the number of history entries in a multiversion b-tree can adversely affect the performance of accessing the current version. This emerges in two situations: linear scan operations and files with a large number of versions. The penalty on lookup operations is reduced by the logarithmic nature of the tree structure, but large numbers of history entries can increase tree depth. Linear scanning of all current entries requires accessing every entry in the tree, which becomes expensive if the number of history entries is high. In both of these cases, it is better to use journal-based metadata.

When lookup of a single entry is common or history access time is important, it is preferable to use multiversion b-trees. Using a multiversion b-tree, all versions of the entry are located together in the tree and have logarithmic lookup time (for both current and history entries), giving a performance benefit over the linear roll-back operation required by journal-based metadata.

9.3.3 Implementation

We have integrated journal-based metadata and multiversion b-trees into a comprehensive versioning file system, called CVFS. CVFS provides comprehensive versioning within our self-securing NFS server prototype. Because of this, some of our design decisions (such as the implementation of a strict detection window) are specific to self-securing storage. Regardless, these structures would be effective in most versioning systems.

9.3.3.1 Overview

Since current versions of file data must not be overwritten in a comprehensive versioning system, CVFS uses a log-structured data layout similar to LFS [Rosenblum91]. Not only does this eliminate overwriting of old versions on disk, but it also improves update performance by combining data and metadata updates into a single disk write.

CVFS uses both mechanisms described in Section 9.3.2. It uses journal-based metadata to version file data pointers and file attributes, and multiversion b-trees to version directory entries. We chose this division of methods based on the expected usage patterns of each. Assuming many versions of file attributes and a need to access them in their entirety most of the time, we decided that journal-based metadata would be more efficient. Directories, on the other hand, are updated less frequently than file meta-data and a large fraction of operations are entry lookup rather than full listing. Thus, the cost of having history entries within the tree is expected to be lower.

Since the only pruning heuristic in CVFS is expiration, it requires a cleaner to find and remove expired versions. Although CVFS's background cleaner is not described in detail here, its implementation closely resembles the cleaner in LFS. The only added complication is that, when moving a data block in a versioning system, the cleaner must update all of the metadata versions that point to the block. Locating and modifying all of this metadata can be expensive. To address this problem, each data block on the disk is assigned a virtual block number. This allows us to move the physical location of the data and only have to update a single pointer within a virtual indirection table, rather than all of the associated metadata.

9.3.3.2 Layout and Allocation

Because of CVFS's log-structured format, disk space is managed in contiguous sets of disk blocks called *segments*. At any particular time, there is a single *write segment*. All data block allocations are done within this segment. Once the segment is completely allocated, a new write segment is chosen. Free segments on the disk are tracked using a bitmap.

As CVFS performs allocations from the write segment, the allocated blocks are marked as either journal blocks or data blocks. Journal blocks hold journal entries, and they contain pointers that string all of the journal blocks together into a single contiguous journal. Data blocks contain file data or metadata checkpoints.

CVFS uses inodes to store a file's metadata, including file size, access permissions, creation time, modification time, and the time of the oldest version still stored on the disk. The inode also holds direct and indirect data pointers for the associated file or directory. CVFS tracks inodes with a unique inode number. This inode number indexes into a table of inode pointers that are kept at a fixed location on the disk. Each pointer holds the block number of the most current metadata checkpoint for that file, which is guaranteed to hold the most current version of the file's inode. The in-memory copy of an inode is always kept up-to-date with the current version, allowing quick access for standard operations. To ensure that the current version can always be accessed directly off the disk, CVFS checkpoints the inode to disk on a cache flush.

Entry Type	Description	Cause
Attribute	Holds new inode attribute information	Inode change
Delete	Holds inode number and delete time	Inode change
Truncate	Holds the new size of the file	File data change
Write	Points to the new file data	File data change
Checkpoint	Points to checkpointed metadata	Metadata checkpoint / Inode change

Table 9-2: Journal entry types. This table lists the five types of journal entry. Journal entries are written when inodes are modified, file data is modified, or file metadata is flushed from the cache.

9.3.3.3 The Journal

The string of journal blocks that runs through the segments of the disk is called the *journal*. Each journal block holds several time-ordered, variably-sized journal entries. CVFS uses the journal to implement both conventional file system journaling (a.k.a. write-ahead logging) and journal-based metadata.

Each journal entry contains information specific to a single change to a particular file. This information must be enough to do both roll-forward and roll-back of the metadata. Roll-forward is needed for update consistency in the face of failures. Roll-back is needed to reconstruct old versions. Each entry also contains the time at which the entry was written and a pointer to the location of the previous entry that applies to this particular file. This pointer allows us to trace the changes of a single file through time.

Table 9-2 lists the five different types of journal entries. CVFS writes entries in three different cases: inode modifications (creation, deletion, and attribute updates), data modifications (writing or truncating file data), and meta-data checkpoints (due to a cache flush or history optimization).

9.3.3.4 Metadata

There are three types of file metadata that can be altered individually: inode attributes, file data pointers, and directory entries. Each has characteristics that match it to a particular method of metadata versioning.

Inode Attributes: There are four operations that act upon inode attributes: creation, deletion, attribute updates, and attribute lookups.

CVFS creates inodes by building an initial copy of the new inode and checkpointing it to the disk. Once this checkpoint completes and the inode pointer is updated, the file is accessible. The initial checkpoint entry is required because the inode cannot be read through the inode pointer table until a checkpoint occurs. CVFS's default checkpointing policy bounds the back-in-time access performance to approximately 150ms as is described in Section 9.3.4.

To delete an inode, CVFS writes a "delete" journal entry, which notes the inode number of the file being deleted. A flag is also set in the current version of the inode, specifying that the file was deleted, since the deleted inode cannot actually be removed from the disk until it expires.

CVFS stores attribute modifications entirely within a journal entry. This journal entry contains the value of the changed inode attributes both before and after the modification. Therefore, an attribute update involves writing a single journal entry, and updating the current version of the inode in memory.

CVFS accesses the current version of the attributes by reading in the current inode, since all of the attributes are stored within it. To access old versions of the attributes, CVFS traverses the journal entries searching for modifications that affect the attributes of that particular inode. Once roll-back is complete, the system is left with a copy of the attributes at the requested point in time.

File Data Pointers: CVFS tracks file data locations using direct and indirect pointers [McKusick84]. Each file's inode contains thirty direct pointers, as well as one single, one double and one triple indirect pointer.

When CVFS writes to a file, it allocates space for the new data within the current write segment and creates a "write" journal entry. The journal entry contains pointers to the data blocks within the segment, the range of logical block numbers that the data covers, the old size of the file, and pointers to the old data blocks that were overwritten (if there were any). Once the journal entry is allocated, CVFS updates the current version of the meta-data to point at the new data.

If a write is larger than the amount of data that will fit within the current write segment, CVFS breaks the write into several data/journal entry pairs across different segments. This compartmentalization simplifies cleaning.

To truncate a file, CVFS first checkpoints the file to the log. This is necessary because CVFS must be able to locate truncated indirect blocks when reading back-in-time. If they are not checkpointed, then the information in them will be lost during the truncate; although earlier journal entries could be used to recreate this information, such entries could expire and leave the detection window, resulting in lost information. Once the checkpoint is complete, a "truncate" journal entry is created containing both a pointer to the checkpointed metadata and the new size of the file.

To access current file data, CVFS finds the most current inode and reads the data pointers directly, since they are guaranteed to be up-to-date. To access historical data versions, CVFS uses a combination of checkpoint tracking and journal roll-back to recreate the desired version of the requested data pointers. CVFS's checkpoint tracking and journal roll-back work together in the following way. Assume a user wishes to read data from a file at time *T*. First, CVFS locates the oldest checkpoint it is tracking with time T_c such that $T_c \geq T$. Next, it searches backward from that checkpoint through the journal looking for changes to the block numbers being read. If it finds an older version of a block that applies, it will use that. Otherwise, it reads the block from the checkpointed metadata.

To illustrate this journal rollback, Figure 9-11 shows a sequence of updates to block 3 of inode 4 interspersed with checkpoints of inode 4. Each block update and inode checkpoint is labeled with the time that it was written. To read block 3 at time $T_1 = 12$, CVFS first reads the checkpoint at time $t = 18$, then reads journal entries to see if a different data block should be used. In this case, it finds that the block was overwritten at time $t = 15$, and so returns the older block written at time $t = 10$. In the case of time $T_2 = 5$, CVFS starts with the checkpoint at time $t = 7$, and then reads the journal entry, and realizes that no such block existed at time $t = 5$.

Directory Entries: Each directory in CVFS is implemented as a multiversion b-tree. Each entry in the tree represents a directory entry; therefore, each b-tree entry must contain the entry's name, the inode number of the associated file, and the time over which the entry is valid. Each entry also contains a fixed-size hash of the name. Although the actual name must be used as the key while searching through the entry blocks, this fixed-size hash allows the index blocks to use space-efficient fixed-size keys.

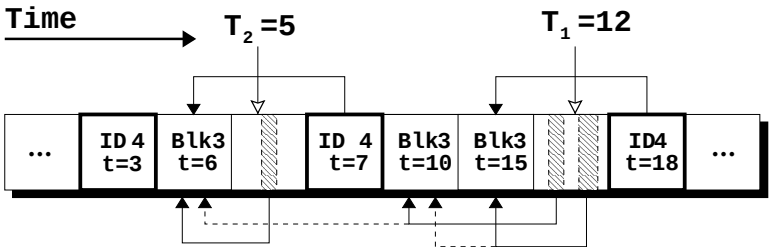


Figure 9-11: Back-in-time access. This diagram shows a series of checkpoints of inode 4 (highlighted with a dark border) and updates to block 3 of inode 4. Each checkpoint and update is marked with a time *t* at which the event occurred. Each checkpoint holds a pointer to the block that is valid at the time of the checkpoint. Each update is accompanied by a journal entry (marked by thin, grey boxes) which holds a pointer to the new block (solid arrow) and the old block that it overwrote (dashed arrow, if one exists).

CVFS uses a full data block for each entry block of the tree, and sorts the entries within it first by hash and then by time. Index nodes of the tree are also full data blocks consisting of a set of index pointers also sorted by hash and then by time. Each index pointer is a ⟨subtree, hash, time-range⟩ tuple, where *subtree* is a pointer to the appropriate child block, *hash* is the name hash of the last entry along the subtree, and *time-range* is the time over which that same entry is valid.

With this structure, lookup and listing operations on the directory are the same as with a standard b-tree, except that the requested time of the operation becomes part of the key. For example, in Figure 9-10a, a lookup of ⟨*C,5*⟩ searches through the tree for entries with name C, and then checks the time-ranges of each to determine the correct entry to return (in this case ⟨*C,4 – 7*⟩). A listing of the directory at time *5* would do an in-order tree traversal (just as in a standard b-tree), but would exclude any entries that are not valid at time *5*.

Insert, remove, and update are also very similar. Insert is identical, with the time-range of the new entry starting at the current time. Remove is an update of the time-range for the requested name. For example, in Figure 9-10b, entry *E* is removed at time *8*. Update is an atomic remove and insert of the same entry name. Also in Figure 9-10b, entry *G* is updated at time *9*. This involves atomically removing the old entry *G* at time *9* (updating the time-range), and inserting entry *G* at time *9* (the new entry ⟨*G,9 – ?*⟩).

Labyrinth Traces					
		Versioned Data	**Versioned Metadata**	**Metadata Savings**	**Total Savings**
Files:	Conventional versioning Journal-based metadata	123.4 GB 123.4 GB	142.4 GB 4.2 GB	97.1%	52.0%
Directories:	Conventional versioning Multiversion b-trees	— —	9.7 GB 0.044 GB	99.6%	99.6%
Total:	Conventional versioning CVFS	123.4 GB 123.4 GB	152.1 GB 4.244 GB	**97.2%**	**53.7%**
Lair Traces					
		Versioned Data	**Versioned Metadata**	**Metadata Savings**	**Total Savings**
Files:	Conventional versioning Journal-based metadata	74.5 GB 74.5 GB	34.8 GB 1.1 GB	96.8%	30.8%
Directo-ries:	Conventional versioning Multiversion b-trees	— —	1.8 GB 0.0064 GB	99.7%	99.7%
Total:	Conventional versioning CVFS	74.5 GB 74.5 GB	36.6 GB 1.1064 GB	**97.0%**	**32.0%**

Table 9-3: Space utilization. This table compares the space utilization of conventional versioning with CVFS, which uses journal-based metadata and multiversion b-trees. The space utilization for versioned data is identical for conventional versioning and journal-based metadata because neither address data beyond block sharing. Directories contain no versioned data because they are entirely a metadata construct.

9.3.4 Evaluation

The objective of this work is to reduce the space overheads of versioning without reducing the performance of current version access. Therefore, our evaluation of CVFS is done in two parts. First, we analyze the space utilization of CVFS. We find that using journal-based metadata and multiversion b-trees reduces space overhead for versioned metadata by over 80%. Second, we analyze the performance characteristics of CVFS. We find that it performs similarly to non-versioning systems for current version access, and that back-in-time performance can be bounded to acceptable levels.

9.3.4.1 Experimental Setup

For the evaluation, we used CVFS as the underlying file system for S4, our self-securing NFS server. S4 is a user-level NFS server written for Linux that uses the SCSI-generic interface to directly access the disk. S4 exports an NFSv2 interface and treats it as a security perimeter between the storage system and the client operating system. Although the NFSv2 specification requires that all changes be synchronous,

S4 also has an asynchronous mode of operation, allowing us to more thoroughly analyze the performance overheads of our metadata versioning techniques.

In all experiments, the client system has a 550 MHz Pentium III, 128 MB RAM, and a 3Com 3C905B 100 MB network adapter. The servers have two 700 MHz Pentium IIIs, 512 MB RAM, a 9 GB 10,000 RPM Quantum Atlas 10K II drive, an Adaptec AIC-7896/7 Ultra2 SCSI controller, and an Intel EtherExpress Pro100 100 Mb network adapter. The client and server are on the same 100 MB network switch.

9.3.4.2 Space Utilization

We used two traces, labeled *Labyrinth* and *Lair*, to evaluate the space utilization of our system. The *Labyrinth* trace is from an NFS server at Carnegie Mellon holding the home directories and CVS repository that support the activities of approximately 30 graduate students and faculty; it records approximately 164 GB of data traffic to the NFS server over a one-month period. The *Lair* trace [Ellard03] is from a similar environment at Harvard; it records approximately 103 GB of data traffic over a one-week period. Both were captured via passive network monitoring.

We replayed each trace onto both a standard configuration of CVFS and a modified version of CVFS. The modified version simulates a conventional versioning system by checkpointing the metadata with each modification. It also performs copy-on-write of directory blocks, overwriting the entries in the new blocks (that is, it uses normal b-trees). By observing the amount of allocated data for each request, we calculated the exact overheads of our two metadata versioning schemes as compared to a conventional system.

Table 9-3 compares the space utilization of versioned files for the two traces using conventional versioning and journal-based metadata. There are two space overheads for file versioning: versioned data and versioned meta-data. The overhead of versioned data is the overwritten or deleted data blocks that are retained. In both conventional versioning and journal-based metadata, the versioned data consumes the same amount of space, since both schemes use block sharing for versioned data. The overhead of versioned metadata is the information needed to track the versioned data. For *Labyrinth*, the versioned metadata consumes as much space as the versioned data. For *Lair*, it consumes only half as much space as the versioned data, because *Lair* uses a larger block size; on average, twice as much data is overwritten with each WRITE.

Journal-based metadata: Journal-based metadata reduces the space required for versioned file metadata substantially. For the conventional system, versioned meta-data consists of copied inodes and sometimes indirect blocks. For journal-based metadata, it is the log entries that allow recreation of old versions plus any checkpoints used to improve back-in-time performance (see Section 9.3.4.3). For both traces, this results in 97% reduction of space required for versioned metadata.

Multiversion b-trees: Using multiversion b-trees for directories provides even larger space utilization reductions. Because directories are a metadata construct, there is no versioned data. The overhead of versioned meta-data in directories is the space used to store the overwritten and deleted directory entries. In a conventional versioning system, each entry creation, modification, or removal results in a new block being written that contains the change. Since the entire block must be kept over the detection window, it results in approximately

9.7 GB of space for versioned entries in the *Labyrinth* trace and 1.8 GB in the *Lair* trace. With multiversion b-trees, the only overhead is keeping the extra entries in the tree, which results in approximately 45 MB and 7 MB of space for versioned entries in the respective traces.

Labyrinth Traces							
		Versioned Data	**Versioned File Metadata**	**Versioned Directories**	**Version Ratio**	**Metadata Savings**	**Total Savings**
Comprehensive:	Conventional	123.4 GB	142.4 GB	9.7 GB			
	CVFS	123.4 GB	4.2 GB	0.044 GB	1:1	97%	54%
On close():	Conventional	55.3 GB	30.6 GB	2.4 GB			
	CVFS	55.3 GB	2.1 GB	0.012 GB	1:2.8	94%	35%
6 minute	Conventional	53.2 GB	11.0 GB	2.4 GB			
Snapshots:	CVFS	53.2 GB	1.3 GB	0.012 GB	1:11.7	90%	18%
1 hour	Conventional	49.7 GB	5.1 GB	2.4 GB			
Snapshots:	CVFS	49.7 GB	0.74 GB	0.012 GB	1:20.8	90%	12%
Lair Traces							
		Versioned Data	**Versioned File Metadata**	**Versioned Directories**	**Version Ratio**	**Metadata Savings**	**Total Savings**
Comprehensive:	Conventional	74.5 GB	34.8 GB	1.79 GB			
	CVFS	74.5 GB	1.1 GB	0.0064 GB	1:1	97%	32%
On close():	Conventional	40.3 GB	6.1 GB	0.75 GB			
	CVFS	40.3 GB	0.57 GB	0.0032	1:2.9	91%	13%
6 minute	Conventional	38.2 GB	3.0 GB	0.75 GB			
Snapshots:	CVFS	38.2 GB	0.36 GB	0.0032	1:11.2	88%	8%
1 hour	Conventional	36.2 GB	2.0 GB	0.75 GB			
Snapshots:	CVFS	36.2 GB	0.26 GB	0.0032	1:15.6	87%	6%

Table 9-4: Benefits for different versioning schemes. This table shows the benefits of journal-based metadata for three versioning schemes that use pruning heuristics. For each scheme it compares conventional versioning with CVFS's journal-based metadata and multiversion b-trees, showing the versioned metadata sizes, the corresponding metadata savings, and the total space savings. It also displays the ratio of versions to le modifications; more modifications per version generally reduces both the importance and the compressibility of versioned metadata.

Other Versioning Schemes: We used the *Labyrinth* and *Lair* traces to compute the space that would be required to track versions in three other versioning schemes: versioning on every file CLOSE, taking systems snapshots every 6 minutes, and taking system snapshots every hour. In order to simulate open-close semantics with our NFS server, we insert a CLOSE call after sequences of operations on a given file that are followed by 500ms of inactivity to that file.

Table 9-4 shows the benefits of CVFS's mechanisms for the three versioning schemes mentioned above. For each scheme, the table also shows the ratio of file versions-to-modifications (e.g., in comprehensive versioning, each modification results in a new version, so the ratio is 1:1). For on-close versioning in the *Labyrinth* trace, conventional versioning requires 55% as much space for versioned metadata as versioned data, meaning that reduction can still provide large benefits. As the versioned metadata to versioned data ratio decreases and as the version ratio increases, the overall benefits of versioned metadata compression drop.

Table 9-4 identifies the benefits of both journal-based meta-data (for "Versioned File Metadata") and multiversion b-trees (for "Versioned Directories"). For both, the meta-data compression ratios are similar to those for comprehensive versioning. The journal-based metadata ratio drops slightly as the version ratio increases, because capturing more changes to the file metadata moves the journal entry size closer to the actual metadata size. The multiversion b-tree ratio is lower because a most of the directory updates fall into one of two categories: entries that are permanently added or temporary entries that are created and then rapidly renamed or deleted. For this reason, the number of versioned entries is lower for other versioning schemes; although multiversion b-trees use less space, the effect on overall savings is reduced.

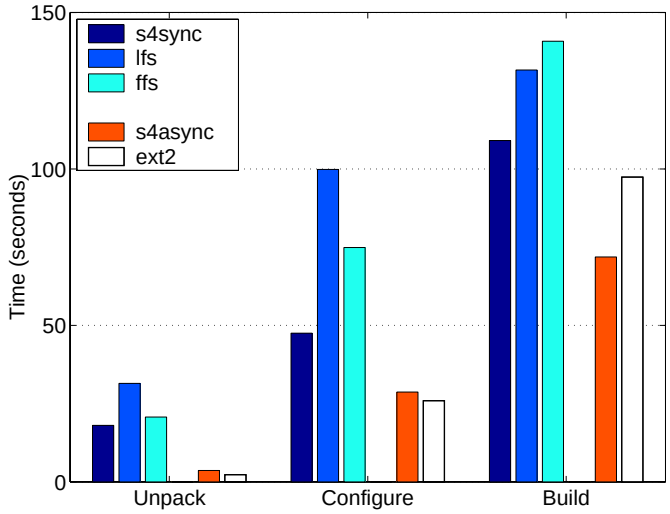


Figure 9-12: SSH comparison. This figure shows the performance of ve systems on the unpack, configure, and build phases of the SSH-build benchmark. Performance is measured in the elapsed time of the phase. Each result is the average of 15 runs, and all variances are under .5s with the exception of the build phases of ffs and lfs which had variances of 37.6s and 65.8s respectively.

9.3.4.3 Performance Overheads

The performance evaluation is done in three parts. First, we compare the S4 prototype to non-versioning systems using several macro benchmarks. Second, we measure the back-in-time performance characteristics of journal-based metadata. Third, we measure the general performance characteristics of multiversion b-trees.

General Comparison: The purpose of the general comparison is to verify that the S4 prototype performs comparably to non-versioning systems. Since part of our objective is to avoid undue performance overheads for versioning, it is important that we confirm that the prototype performs reasonably relative to similar systems. To evaluate the performance relationship between S4 and non-versioning systems, we ran two macro benchmarks designed to simulate realistic workloads.

For both, we compare S4 in both synchronous and asynchronous modes against three other systems: a NetBSD NFS server running FFS, a NetBSD NFS server running LFS, and a Linux NFS server running EXT2. We chose to compare against BSD's LFS because it uses a log-structured layout similar to S4's. BSD's FFS and Linux's EXT2 use similar, more "traditional" file layout techniques that differ from S4's log-structured layout. It is not our intent to compare a LFS layout against other layouts, but rather to confirm that our implementation does not have any significant performance anomalies. To ensure this, a small discussion of the performance differences between the systems is given for each benchmark.

Each of these systems was measured using an NFS client running on Linux. Our S4 measurements use the S4 server and a Linux client. For "Linux," we run RedHat 6.1 with a 2.2.17 kernel. For "NetBSD," we run a stock NetBSD 1.5 installation.

To focus comparisons, the five setups should be viewed in two groups. BSD LFS, BSD FFS, and S4-sync all push updates to disk synchronously, as required by the NFSv2 specification. Linux EXT2 and S4-async do not; instead, updates are made in memory and propagated to disk in the background.

SSH-build [Seltzer00] was constructed as a replacement for the Andrew file system benchmark [Howard88]. It consists of 3 phases: The unpack phase, which unpacks the compressed tar archive of SSH v1.2.27 (approximately 1 MB in size before decompression), stresses metadata operations on files of varying sizes. The configure phase consists of the automatic generation of header files and Makefiles, which involves building various small programs that check the existing system configuration. The build phase compiles, links, and removes temporary files. This last phase is the most CPU intensive, but it also generates a large number of object files and a few executables. Both the server and client caches are flushed between phases.

Figure 9-12 shows the SSH-build results for each of the five different systems. As we hoped, our S4 prototype performs similarly to the other systems measured.

LFS does significantly worse on unpack and configure because it has poor small write performance. This is due to the fact that NetBSD's LFS implementation uses a 1 MB segment size, and NetBSD's NFS server requires a full sync of this segment with each modification; S4 uses a 64kB segment size, and sup-

ports partial segments. Adding these features to NetBSD's LFS implementation would result in performance similar to S4. FFS performs worse than S4 because FFS must update both a data block and inode with each file modification, which are in separate locations on the disk. EXT2 performs more closely to S4 in asynchronous mode because it fails to satisfy NFS's requirement of synchronous modifications. It does slightly better in the unpack and configure stages because it maintains no consistency guarantees, however it does worse in the build phase due to S4's segment-sized reads.

Postmark was designed to measure the performance of a file system used for electronic mail, netnews, and web based services [Katcher97]. It creates a large number of small randomly-sized files (between 512 B and 9 KB) and performs a specified number of transactions on them. Each transaction consists of two sub-transactions, with one being a create or delete and the other being a read or append. The default configuration used for the experiments consists of 20,000 transactions on 5,000 files, and the biases for transaction types are equal.

Figure 9-13 shows the Postmark results for the five server configurations. These show similar results to the SSH-build benchmark. Again, S4 performs comparably. In particular, LFS continues to perform poorly due to its small write performance penalty caused by its interaction with NFS. FFS still pays its performance penalty due to multiple updates per file create or delete. EXT2 performs even better in this benchmark because the random, small file accesses done in Postmark are not assisted by aggressive prefetching, unlike the sequential, larger accesses done during a compilation; however, S4 continues to pay the cost of doing larger accesses, while EXT2 does not.

Journal-based Metadata: Because the metadata structure of a file's current version is the same in both journal-based metadata and conventional versioning systems, their current version access times are identical. Given this, our performance measurements focus on the performance of back-in-time operations with journal-based metadata.

There are two main factors that affect the performance of back-in-time operations: checkpointing and clustering. Checkpointing refers to the frequency of metadata checkpoints. Since journal roll-back can begin with any checkpoint, CVFS keeps a list of metadata checkpoints for each file, allowing it to start roll-back from the closest checkpoint. The more frequently CVFS creates checkpoints, the better the back-in-time performance.

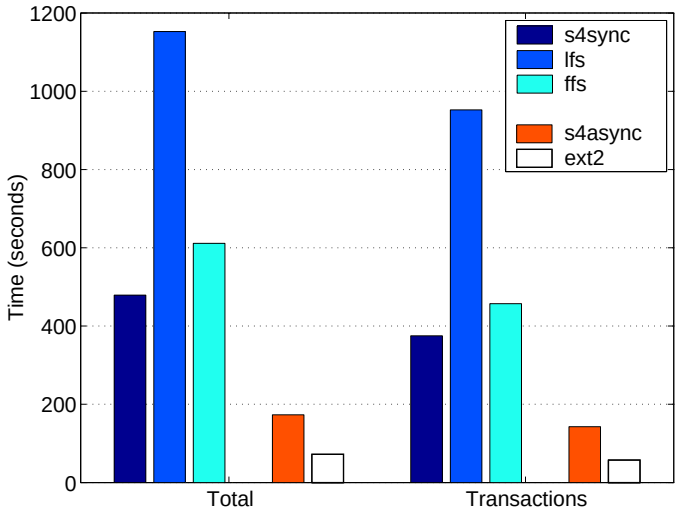


Figure 9-13: Postmark comparison. This figure shows the elapsed time for both the entire run of postmark and the transactions phase of postmark for the five test systems. Each result is the average of 15 runs, and all variances are under 1.5s.

Clustering refers to the physical distance between relevant journal entries. With CVFS's log-structured layout, if several changes are made to a file in a short span of time, then the journal entries for these changes are likely to be clustered together in a single segment. If several journal entries are clustered in a single segment together, then they are all read together, speeding up journal rollback. The "higher" the clustering, the better the performance is expected to be.

Figure 9-14 shows the back-in-time performance characteristics of journal-based metadata. This graph shows the access time in milliseconds for a particular version number of a file back-in-time. For example, in the worst-case, accessing the 60th version back-in-time would take 350ms. The graph examines four different scenarios:

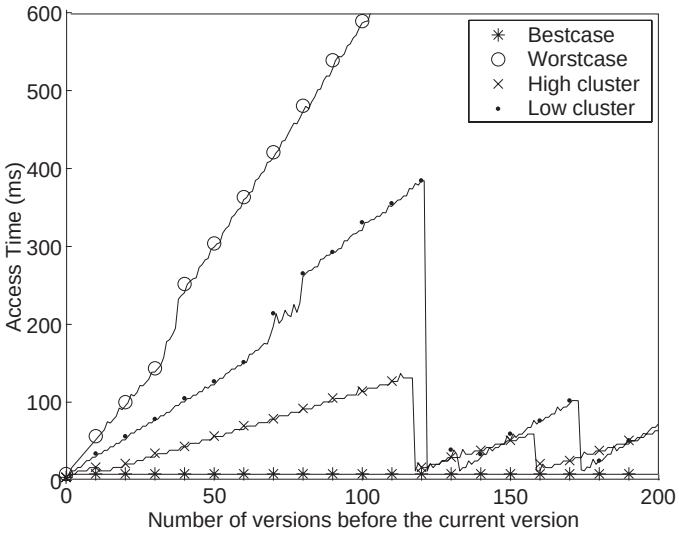


Figure 9-14: Journal-based metadata back-in-time performance. This figure shows several potential curves for back-in-time performance of accessing a single 1KB file. The worst-case is when journal roll-back is used exclusively, and each journal entry is in a separate segment on the disk. The best-case is if a checkpoint is available for each version, as in a conventional versioning system. The high and low clustering cases are examples of how checkpointing and access patterns can affect back-in-time performance. Both of these cases use random checkpointing. In the "high cluster" case, there is an average of 5 versions in a segment. In the "low cluster" case, there is an average of 2 versions in a segment. The cliffs in these curves indicate the locations of checkpoints, since the access time for a checkpointed version drops to the best-case performance. As the level of clustering increases, the slope of the curve decreases, since multiple journal entries are read together in a single segment. Each curve is the average of 5 runs, and all variances are under 1ms.

best-case behavior, worst-case behavior, and two potential cases (one involving low clustering and one involving high clustering).

The best-case back-in-time performance is the situation where a checkpoint is kept for each version of the file, and so any version can be immediately accessed with no journal roll-back. This is the performance of a conventional versioning system. The worst-case performance is the situation where no checkpoints are kept, and every version must be created through journal roll-back. In addition there is no clustering, since each journal entry is in a separate segment on the disk. This results in a separate disk access to read each entry. In the high clustering case, changes are made in bursts, causing journal entries to be clustered together into segments. This reduces the slope of the back-in-time performance curve. In the low clustering case, journal entries are spread more evenly across the segments, giving a higher slope. In both the low and high clustering cases, the points where the performance drops back to the best-case are the locations of checkpoints.

Using this knowledge of back-in-time performance, a system can perform a few optimizations. By tracking checkpoint frequency and journal entry clustering, CVFS can predict the back-in-time performance of a file while it is being written. With this information, CVFS bounds the performance of the back-in-time operations for a particular file by forcing a checkpoint whenever back-in-time performance is expected to be poor. For example, in Figure 9-14, the high-clustering case keeps checkpoints in such a way as to bound back-in-time performance to around 100ms at worst. In our S4 prototype, we bound the back-in-time performance to approximately 150ms. Another possibility is to keep checkpoints at the point at which one believes the user would wish to access the file. Using a heuristic such as in the Elephant FS [Santry99] to decide when to create file checkpoints might closely simulate the back-in-time performance of conventional versioning.

Multiversion B-trees: Figure 9-15 shows the average access time of a single entry from a directory given some fixed number of entries currently stored within the directory (notice the log scale of 13 the x-axis).

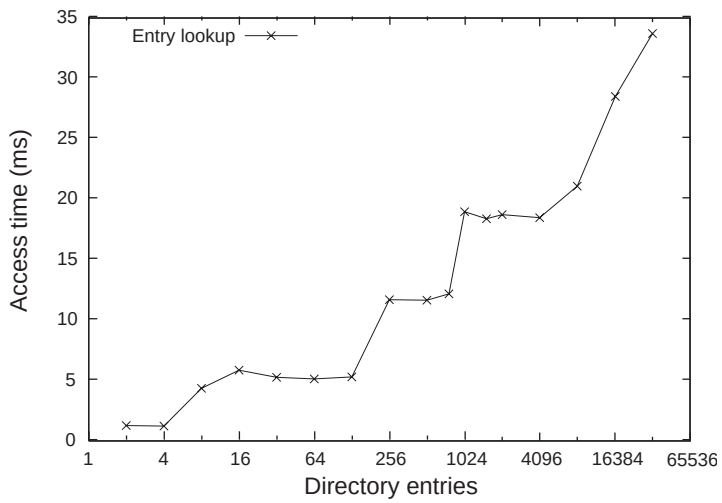


Figure 9-15: Directory entry performance. This figure shows the average time to access a single entry out of the directory given the total number of entries within the directory. History entries affect performance by increasing the effective number of entries within the directory. The larger the ratio of history entries to current entries, the more current version performance will suffer. This curve is the average of 15 runs and the variance for each point is under .2ms.

To see how a multiversion b-tree performs as compared to a standard b-tree, we must compare two different points on the graph. The point on the graph corresponding to the number of current entries in the directory represents the access time of a standard b-tree. The point on the graph corresponding to the combined number of current and history entries represents the access time of a multiversion b-tree. The difference between these values is the lookup performance lost by keeping the extra versions.

Using the traces gathered from our NFS server, we found that the average number of current entries in a directory is approximately 16. Given a detection window of one month, the number of history entries is less than 100 over 99% of the time, and between zero and five over 95% of the time. Since approximately 200 entries can fit into a block, there is generally no performance lost by keeping the history. This block-based performance explains the stepped nature of Figure 9-15.

9.3.4.4 Summary

Our results show that CVFS reduces the space utilization of versioned metadata by more than 80% without causing noticeable performance degradation to current version access. In addition, through intelligent checkpointing, it is possible to bound back-in-time performance to within a constant factor of conventional versioning systems.

9.3.5 Metadata Versioning in Non-Log-Structured Systems

Most file systems today use a layout scheme similar to that of BSD FFS rather than a log-structured layout. Such systems can be extended to support versioning relatively easily; Santry et al. [Santry99] describe how this was done for Elephant, including tracking of versions and block sharing among versions. For non-trivial pruning policies, such as those used in Elephant and CVFS, a cleaner is required to expire old file versions. In an FFS-like system, unlike in LFS, the cleaner does not necessarily have the additional task of coalescing free space.

Both journal-based metadata and multiversion b-trees can be used in a versioning FFS-like file system in much the same way as in CVFS. Replacing conventional directories with multiversion b-trees is a self-contained change, and the characteristics should be as described in the paper. Replacing replicated metadata versions with journal-based metadata requires effort similar to adding write-ahead logging support. Experience with adding such support [Hagmann87, Seltzer00] suggests that relatively little effort is involved, little change to the on-disk structures is involved, and substantial benefits accrue. If such logging is already present, journal-based metadata can piggyback on it.

Given write-ahead logging support, journal-based meta-data requires three things. First, updates to the write-ahead log, which are the journal entries, must contain enough information to roll-back as well as roll-forward. Second, the journal entries must be kept until the cleaner removes them; they cannot be removed via standard log checkpointing. This will increase the amount of space required for the log, but by

much less than the space required to instead retain metadata replicas. Third, the metadata replica support must be retained for use in tracking metadata version checkpoints.

With a clean slate, we think an LFS-based design is superior if many versions are to be retained. Each version committed to disk requires multiple updates, and LFS coalesces those updates into a single disk write. LFS does come with cleaning and fragmentation issues, but researchers have developed sufficiently reasonable solutions to them to make the benefits outweigh the costs in many environments. FFS-type systems that employ copy-on-write versioning have similar fragmentation issues.

9.4 Conclusions

Self-securing storage ensures data and audit log survival in the presence of successful intrusions and even compromised host operating systems. Experiments with the S4 prototype show that self-securing storage devices can achieve performance that is comparable to existing storage appliances. In addition, analysis of recent workload studies suggest that complete version histories can be kept for several weeks on state-of-the-art disk drives.

Our work on versioning shows that journal-based metadata and multiversion b-trees address the space-inefficiency of conventional versioning. Integrating them into the CVFS file system has nearly doubled the detection window that can be provided with a given storage capacity. Further, current version performance is affected minimally, and back-in-time performance can be bounded reasonably with checkpointing.

10 PROGRAM ANALYSIS FOR RELIABLE INFORMATION SYSTEMS

10.1 Introduction

As programmers, we often overlook error checking, as we believe that the various errors are just inconceivable and do not need to be checked or are overwhelmed with the task of identifying all possible error situations.

Unhandled error conditions lead to potential software failures when the underlying system cannot satisfy our requests. Programmers alone cannot thoroughly account for all error conditions, as it is a non-trivial task with a lack of tool support. For instance, the error checking code in the C programming language standard library is diverse. There is no one programming method that accounts for all error identification. Some types of errors require checks to be performed on a value that has been received from a function call, some require a value check before further computation, and some types check against an associated value. While error checking is difficult, it is not always absent. When it is absent, it is then necessary for the system to adapt itself during a system error, trading computational or other resources for robustness.

Storage subsystems will make reasonable attempts to provide data when possible. Transient problems such as network congestion or outages, heavily loaded systems, or denial-of-service attacks can lead to failure-like situations where it is not possible for the storage subsystem to perform the entire requested operation. Yet, since the problem is only transient, it may still provide a partial result. The semantics of application calls into storage subsystems make allowances for cases where not all data is available, but where failure conditions do not exist. A true failure condition for a storage subsystem is a request for data where no data is available, such as a read past the end of a file, and is different from a situation in which data is available, but is currently not accessible.

In the case of network congestion or a heavily loaded storage node, the situation is not that the data cannot be obtained, but that a timeout has occurred in the request for data. This timeout is placed by the network communication protocol in the operating system so that the system returns to the calling application within a reasonable amount of time. After a timeout, the storage subsystem returns control to the calling code. The storage subsystem has several alternative ways of dealing with a timeout when requesting data. It can re-request the data again, assuming that if a transient failure has occurred it may be able to retrieve the data in a second attempt. There may also be several replicas of the data on the network or other means of obtaining the data, so it may be possible to contact a different location.

In a very heavily loaded system it may not useful to request information again or contact every possible source, thus adding to the network congestion. It is especially inadvisable to attempt alternative solutions when code provided by the programmer exists expressly to handle the case of limited data availability. The solution provided by the programmer would be optimal yet specific to the application. However, in the absence of code explicitly put in place to handle less than full data availability, it is possible to trade performance (a slowdown of the response to the call to the storage subsystem) for robustness using several of the alternatives described earlier. We present a system to coordinate the program characteristics between the program and system.

The chapter is organized as follows: In section 10.2, we identify our fundamental understanding of exception handling. Related work is described in section 10.3. In section 10.4, we discuss the results of the design and implementation of four software exception injection (SWEI) systems. In section 10.5 we will outline the key points that must be understood to build an exception injection framework. Section 10.6 discusses the use of exception analysis with survivable storage alternatives. We will then present the observations that arose out of our exception injection experimentations, as well as our conclusions and the directions in which we see future work proceeding. For its contents, the chapter draws primarily work published the following papers: “Testing the portability of desktop applications to a networked embedded system” [Bigrigg01], “The Set-Check-Use Methodology for Detecting Error Propagation Failures in I/O

Routines" [Bigrigg02a], and "Robustness Hinting for Improving End-to-End Dependability" [Bigrigg02b].

10.2 Exception Handling Identification Methodology

Exceptions can be used to provide additional information from a callee to a caller. This information may be the identification of an error condition, or it may simply be informative [Goodenough75, Lang98, Lee83, Melliar-Smith77]. For example, the C standard library fopen() function will return a file handle upon successful completion, and will return a NULL (typically a zero) to signify an error [Plauger92]. However, the C fread() function will return the number of bytes read. The typical case is that the return value matches the number of bytes requested. The return value may be less than the number requested, informing the caller that less than the number of bytes remained in the file. If the value is a zero, it means that the end of file has been reached and that there is no data left. If the number is a negative value, an error has occurred. We will classify an exception as informational data given from the callee to the caller such that the information returned is important to the further execution of the application. For example, the C math library sin() function, which computes the sine of a value, has a return value. This return value may be ignored. Of course it may seem to be ridiculous to compute a value and not use it, but the ignoring of the value of sin(), unlike the return value of fread(), does not effect the robustness of the application.

Exceptions are available at the language level and at an application level. Language exceptions are built into the language and the standard libraries such as the I/O processing done by the operating system. Application exceptions are part of the return message passed from one component to another.

Language exceptions are, of course, based on the programming language. The language C uses overloaded return values to identify an exception. In addition it uses an errno global variable to further explain the error. Object-oriented languages such as Java, C++, and Ada, have runtime exception mechanisms. With language enabled exception handling, it is much easier to determine if they exception was acknowledged or ignored.

Application exceptions are built into the protocol of communication between application components. This is necessary when communication is done between a device and the host computer in the case of the small computer system interface (SCSI) protocol. And also for the communication mechanism between computers in the case of remote procedure calls (RPC), remote method invocation (RMI), or the common object request broker architecture (CORBA). It is not universal that all communication protocols express exceptions. For instance, the HTTP protocol does not have exceptions as a part of its protocol.

Both language-based exceptions and application-based exceptions must be dealt with through a recovery mechanism or by reporting the error condition back to the user. Missing exception handling does not lead to catastrophic crashes, but cause silent failures and infinite loops. Part of the process of building an exception injection system is to codify the exceptions. It should be noted that part of the problem in exception handling might be the arbitrary nature to the way exceptions are expressed. In POSIX, the fopen() function returns a NULL (a zero in most systems) as its failure exception and the fclose() function returns an EOF (a –1 in most systems) as its failure exception. The lack of an orthogonal approach to exceptions may be a major cause in the misidentification of exceptions.

Software Fault Detection

The interface to a local file system is identical to that of a distributed file system though the potential for failure is greatly different. In a local file system failures are catastrophic. If the hard drive or other local storage device fails, it is often at the end of the device's usable lifetime. However, failures in a distributed file system are more common resulting from remote storage devices that are unreachable or overloaded due to network partitioning, poor load balancing, or a denial of service attack. There are assumptions about the reliability of the system built into the application but is not often expressed. Yet unhandled error conditions lead to potential software failures when the underlying system cannot satisfy our requests and

the application was built assuming that it can. We present a methodology based upon program static analysis to track the propagation of error reporting in order to determine the assumptions used when the software was created.

We present a methodology that will detect in source code a robustness failure where an error could occur and where there is no mechanism is in place to handle the error. Many programmers fail to incorporate error checking in specific classes of I/O operations and rely on certain assumptions such as file output is always guaranteed. It is this absence of error checking that we intend to detect with our methodology. Our approach to uncovering these situations combines an augmented data flow analysis with the semantics of the I/O error reporting.

We describe the details of the methodology showing how to augment traditional compiler data flow analysis to find structurally, within the application, code that would be used to perform error checking. In addition we describe how it is used to ensure the correctness of the error checking.

Error Reporting in POSIX I/O Routines

Errors are reported in POSIX I/O routines using out-of-range values. The return values of these routines are either a useful result (upon successful completion of the call) or an indication of the error that occurred (upon and unsuccessful completion). For instance, the successful return of the fopen call is a handle to a file. The range of values for a file handle is an unsigned integer greater than zero. A zero, also referred to as NULL, is then used to report that the file system was unable to open the file. The return of a fread call uses out-of-range values to transmit not only an error condition, but also specifies an end of file condition as well. The return identifies the number of bytes that have actually been read. The fread call, like all data buffer operations, will read up to but no more than the number of bytes that have been requested. A return of 0 does not signify an error condition, just that no data is currently accessible such as at the end of a file. It is a negative return value that signifies an error condition. Since a single value can potentially be both an error condition and also a valid result, it is not until tested that we know. Just like Schroedinger's cat, you cannot tell what the value is until you examine it. When writing a program, we have to assume that both outcomes are likely and cannot only assume one.

The values that specify an error condition are based on the I/O routine itself. An examination of the C standard I/O library [Cristian95] shows the behavior of I/O function calls upon an error condition:

- Functions that return pointers use a NULL to designate an error condition: `fmpfile, fopen, freopen, fgets.`
- Functions that use EOF as an error condition: `fclose, fgetc, getchar, putchar, puts, ungetc.`
- Functions that use a non-zero for an error condition: `remove, rename.`
- Functions that use a negative number for an error condition: `fputs, fgetpos, fseek, fsetpos, fprintf, fscanf, print, sprintf, sscanf, vfprintf, vsprintf, fputc, fputs, gets, putc, fread, fwrite.`
- Functions that use a -1 for an error condition: `ftell.`

Only the data buffer operations, (`fprintf, fscanf, print, sprintf, sscanf, vfprintf, vsprintf, fputc, fputs, gets, putc, fread, fwrite`) overload the return with three potential values.

In addition, we must identify the result value. The result is the value achieved upon successful completion of the call and may be passed through a return or through an argument. The buffer operations have not only the result in the return but also an argument, a buffer, which is also a result. Not only is it important

to distinguish the error from the result in the return, but also it is important to acknowledge the error before using the buffer contents. Error checking needs to occur before the use of any result values.

Identification of error checking

Correct error checking associated with an I/O routine must occur between the set (called a definition or simply *def*) of the potential error value and *use* of a result value or values along all possible paths of execution. For instance, the C code in Figure 10-1 could lead to a program crash, while the code in Figure 10-2 uses program logic to safeguard against a possible error condition.

```
fin = fopen("foo","r");
fread (fin, sizeof(int), 10, buf);
```

Figure 10-1: Code that may lead to a failure.

```
fin = fopen("foo","r");
if (fin != NULL) {
      fread(fin, sizeof(int), 10, buf);
}
```

Figure 10-2: Program logic guards against a possible unsuccessful result.

We augment traditional data flow analysis to identify missing error checking. Data flow analysis is a traditional technique used by compilers during the optimization phase as a tool to guarantee the correctness of program transformations. Value chains, called *def-use chains*, are identified between the definition of a value and the places the value is used. Data analysis is performed on values and not on variables. Figure 10-3 shows how a value chain is formed, not dependent on the name of the variable, but on the instance of a value in a variable.

```
a = 3; /* def of a1 */
b = a +5; /* use of a1 , def of b1 */
a = 8; /* def of a2 */
```

Figure 10-3: Formation of a value chain.

We augment the def-use chains to additionally include the *check* of a value. We define a *check* as a *use* of a value that additionally falls within the expression of a conditional statement. There is already a large body of work on the mechanisms for computing def-use chains. [Dasgupta99] The conditional is a guard against the incorrect usage of the result value. The conditional expression that performs an error guard may be part of any conditional structure including *if* and *if-else* statements as well as while and repeat loops as shown in Figure 10-4.

```
n = fread (fin, sizeof(int), 1, buf);
while (n > 0) {
      k += buf[0];
      n = fread (fin, sizeof(int), 1, buf);
}
```

Figure 10-4: Formation of a value chain.

While set-check-use is a straightforward approach there are a few issues to incorporate into our methodology. The error value and the result values are not bound to a specific variable as shown in Figure 10-5. These values can be assigned to other variable or even modified. In these cases, we need to track the values to make sure that the *use* of the result values does not come before the *check* of the error values.

```
a = fopen (“foo.txt”,”r”);
b = a;
if (b != NULL) {
      n = read(a, sizeof(int), 1, buf);
}
```

Figure 10-5: Analysis based on values not variables.

Also, the use of the result value does not only have to exist within the body of the conditional, but the conditional may be used to reset the result variable as in Figure 10-6. Again this involves tracking the values through all execution paths. Once the tracked value is overridden with another value, the tracking of the previous value stops along that path of execution.

```
a = fopen (“foo.txt”, “r”);
if (a == NULL) {
      a = stdin;
}
```

Figure 10-6: Resetting a value for protection.

We know that there is a check, but that does not mean that the expression will accurately identify an error situation.

Finally, in order to determine if the check is valid we must examine the conditional expression. This will be explained in the remainder of this section.

Another aspect to detecting missing robustness checks is the use of error information from the language, as listed in the previous section, to guide the *set-check-use* approach by identifying what value identifies the error. The error propagation information provides a heuristic approach similar to error classification schemes [Chakrabarti02]. An example is given to show how semantic information would drive the *def-check-use* analysis. In the case of file opening as shown in Figure 10-7, the def, check, and use locations use the same value for the analysis. Between the *def* and *use* of *a*, there should be a *check* of *a*.

```
a = fopen("foo","r"); /* def of a */
if (a != NULL) /* check of a */
      fread (a, sizeof(int), 10, buf); /*use of a */
```

Figure 10-7: Def-Check-Use of the Same Value.

In an `fopen` call, the return holds the error value. A NULL return value designates an error condition. The *def*, *check*, and *use* is to use the same value, *a*, which is the value returned from the fopen call. We acknowledge that the validation of the check may not be possible to determine statically.

Analysis of a Simple Program

The wc program is part of the GNU textutils collection of programs. Its purpose is to count the number of lines, words, and characters in a file or files identified on the command line. The v2.0 program consists of 371 lines or 118 lines of code. It can be identified to have been in use for the past 16 years (from 1985 to 2001). It was written in C and consists of four functions. There were no instances of control flow issues where error checking only exists in a subset of execution paths. A hand analysis of the main source pro-

gram was performed using the methodology presented to detect a failure to check for an error condition. The results are summarized in Table 10-1.

Routine	Total	Checked	Unchecked
fprintf	1	0	1
Printf	7	0	7
Puts	1	0	2
putchar	1	0	2
fstat	1	1	0
lseek	2	2	0
read	3	3	0
open	1	1	0
close	2	2	0
setlocale	1	0	1
SUMMARY	**20**	**9**	**11**

Table 10-1. Hand Analysis of the wc program

A simple calculation of the number of checks that should be performed against the number of checks actually performed results in a 45% reliability rating. The usefulness of this rating is not promoted as it does not reflect the frequency of each call, but can be included as a guide to understand the program behavior.

The major assumption that was made by the *wc* program is that all output is guaranteed to succeed.

10.3 Related Work

We are not developing an explicit software fault injection (SWFI) system for I/O [Miller90]. A fault injection system for I/O would be to remove the data device or flip bits, to stress the system based on known ways the system fails. Fault injection will perturbate the data or the system in an attempt to raise an error or exception. Exception injection explicitly raises the exception directly. Our SWEI does not contain fault models, which identify the faults upon which exceptions may be introduced into the system. Our injection is based on a listing of all the possible exceptions that may be raised in the course of execution. A fault model may be necessary for non-I/O based systems that contain function calls with side effects and is mentioned in the future works section.

There are two approaches to the detection of missing exception handling. The first one is static analysis through the use of compiler techniques along with formal methods. These systems do not provide explicit validation of exception handling; they provide the mechanisms to allow for exception handling validation. The second approach is dynamic analysis or testing, often in a similar form to fault injection.

Static Analysis

The LcLint [Evans94] system is based on the well-known lint [Johnson78] tool. Lint was developed to identify portability problems in an application. For instance, lint will identify if the application is performing operations that depend upon the size of an integer. The size of an integer is not consistent among architectures. The LcLint system uses the similar idea of heuristics based on common programming problems to detect potential problems in source code. LcLint along with the other static analysis tools presented are extensible to allow for programmer-defined rules.

The CHIC [Chakrabarti02] and mc [Engler01, Engler00] systems are formal-method systems that provide a framework from which a programmer can specify rules to determine if there are problems within source code. Both systems are shown to be very effective at identification of incorrect programming behavior. It is necessary to be able to codify the incorrect behavior into rules.

The static analysis tools presented all have the capability to help identify the lack of correct exception handling. These systems presuppose that programmers missed a location where exception handling should be present or that perhaps the programmers handled the wrong exception. We have proposed our own methodology [Bigrigg02] for the identification of missing exception handling which could be integrated into any of these frameworks presented.

Static analysis tools are effective ways to identify missing error handling, but are not effective at validating the code used to handle the exceptions.

Dynamic Analysis

Two systems that do fault injection of I/O are part of the Recovery-Oriented Computing group of Berkeley and Stanford. The FIG system [Broadwell02] does fault injection and is similar to our FlakyPOSIX system. FIG will modify the runtime library to insert a fault injection system to test the application's recovery mechanisms. Similar to FIG is the RAID testing [Brown00]. Faults were also injected, this time into a RAID system to determine its ability to handle errors.

Fuzz [Miller90] is a fault injection system that returns random data back to the application in an attempt to see how robust the I/O processing is. This is a true I/O fault injection system as it perturbates the incoming data to raise an exception.

While fault injection systems will test the exception handling code, it does not provide a means of exercising a single exception handler. This is necessary to integrate exception handling testing into the software development cycle.

10.4 FlakyIO Family

The Flaky Family consists of several systems that have been developed using the Flaky exception injection model. It is a model and not an integrated system, simply because the components are so diverse that it was not possible to make one single flaky system that encompassed everything.

The FlakyIO family consists of FlakyPOSIX, FlakyPalm, FlakyDisk, and FlakyNet. Overall the results show that every system has application instances that do not do correct exception handling. Due to space limitations we do not go into detailed analysis of the results, but highlight the most salient features, which impacted greatly our understanding of how to design an exception injection system.

FlakyPOSIX

Our first generation exception injection system was originally called FlakyIO. The name was changed to FlakyPOSIX as we began to develop other I/O-based exception injection systems. It consisted of a fault exception engine located on the application side based on a C source-code modification tool. The FlakyPOSIX tool was first applied to the GNU binutils and the GNU textutils.

Exceptions were raised based on function call. We reported the results of the running of each of the applications emulating transient exceptions based on the first call to the function. The applications handled the exception correctly (HC), handled it incorrectly (HI), or failed silently (S). Most applications were also written to report their errors via a printf type call without checking to see if the error was reported properly (S2).

Original observations concluded that some applications do and some applications do not do proper error testing. We have explored the ways in which the programs were developed in more depth and have discovered a greater complexity which leads us to believe it would be very difficult to codify rules for the behavior of proper exception handling. For instance, cksum is reported to silently fail its fread() call. The cksum program checks to see if the result of the fread() call is less than or equal to zero. A zero signifies the end of file, while a negative value signifies an error. We have marked this program as not handling the exception. This is only partially correct, as cksum will also make an additional call after the fread() to

feof() to determine if the file is at the end of the file. While cksum will catch a permanent failure of the disk, it will not be able to handle transient errors.

FlakyPOSIX mySQL. The next thing we did was to apply the FlakyPOSIX to a mySQL database. We used the same configuration as we did for the GNU textutils: exception engine on the application side, function call boundary, and we raised a transient exception on the first call to a function. The results were initially not what we had expected. Instead of a mixed bag of results similar to the textutils, we found that no matter what the test, the result was the same "good-bye." The C POSIX I/O routines had been wrapped to exit based on error conditions.

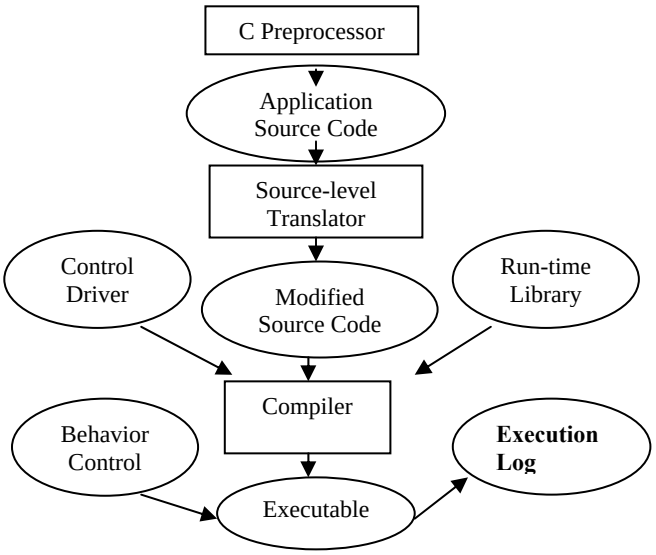


Figure 10-8. Test harness phases

We have grown to like the "good-bye" message. It is a simple and graceful exit of the application. It is an acceptable handling of the exception. We did not examine the code to determine if the database was left in a correct state.

Test Harness

We first developed a testing harness to allow us to selectively manipulate the behavior of the function calls in the application source code. The test harness modifies the application at the source code level. It consists of a source-level translator, a run-time library, and a control driver.

The source-level translator is the front end of a C compiler. The C preprocessor is first run on all source code before being passed to the translator as shown in Figure 10-8. The translator builds a parse tree to be used to modify the source code and add in conditional statements to give control over specific function calls [Alexandrov97]. For example a normal fread function call in the sort text utility looks like:

```
cc = fread (buf ->buf +buf ->used, 1,
      buf->alloc-1-buf->used, fp ));
```

At the point of the function call a conditional is inserted around the function call to give control to the driver over the execution of the fread function. This is shown below:

```
cc =(control_fread ( 14262 )
      ?paris_fread ()
      :fread (buf->buf+buf->used, 1,
            buf->alloc-1-buf->used, fp));
```

The control_<function>(<line number>) implements a call to the driver to determine whether to execute or fail the function. It takes in the line number where the function is found in the code to help in analysis of the results when several function calls are being made. The control driver can fail functions on a specific instance of a call or for all call instances. For example, fread can be failed at a given point in a program by returning the error code rather than executing the function. A behavior control input file allows for functions to be specified when to fail: always fail, fail consecutively, or fail on a certain line number.

A log is generated that tracks the progress of the application and tracks the calling pattern of the application.

The paris_<function> returns the appropriate error value based on the specified function. For example, the C standard I/O library fread call will return up to but no more than the number of bytes that have been

requested. A return value of 0 does not signify an error condition, just that no data is currently available [Bigrogg01]. A return value of –1 signifies an error and is what will be returned by paris_fread. This information is then passed to the calling code to be handled. It is then the calling code's responsibility to handle the problem.

The use of program analysis to identify potential sources of problems has been previously presented in such systems as lint [Bigrigg02] that uses the approach to check for common portability errors. All problems are reported to the application programmer to take appropriate action. The errors are those typical when an assumption is made about the processor architecture. LcLint [Broadwell02] extends the lint system to allow for specifications allowing a more customizable portability check.

There are also tools that attempt to classify the robustness of an application and present mechanisms for the automatic generation of run-time test coverage. For example, the Ballista project [Brown00] stresses the API of a module by making calls to a module using a series of extreme values. The behavior of the module is examined and reported to the application programmer.

Our system focuses on the proper handling of error conditions using a combination of program analysis and run-time testing.

Test Cases

We applied the testing harness to a specific class of applications that are I/O based and use the C standard I/O library. In the case of networked embedded systems, especially that of a mobile environment, the situation is not always that the data cannot be obtained, but may be that a timeout has occurred in the request for data. This timeout is placed by the network communication protocol in the operating system so that the system returns to the calling application within a reasonable amount of time, but also results in the generation of an error.

Our error handling testing was performed on the GNU binutils and textutils. There are 26 text utility and 15 bin utility applications written by several authors. They are all written in C and rely heavily on the C standard I/O library. They have been ported to numerous platforms and are a widely accepted and used set of utilities. Our experiments were performed using the Linux platform.

The test harness was applied as described in section 10.4.2.1. The source files for the utilities as well as supplemental libraries were run through the C preprocessor, test harness source translator, and then compiled. We first ran the applications in their normal operating mode and then in our test mode. As displayed in the chart, several of the applications were unable to be successfully modified by the test harness. These utilities were not built due to incompatibilities between cpp and our translator. The majority of utilities were built successfully and are a significant representation of the packaged source code.

Results

After analyzing all of the utilities for how they handle different error situations, four different failure categories became apparent. These four are: handles correctly (HC), handles incorrectly (HI), silent failure (S), and silent failure (S2) when failed in conjunction with another function.

Handles correctly (HC) is when an application recognizes that an error has occurred, an error return value has been acknowledged by the calling function and an appropriate error display is given. This is correct behavior.

Handles incorrectly (HI) is similar to handles correctly in as much as the application acknowledges that an error has occurred and an error return value has been received, but an accurate error display is not given.

Text Utilities	fread()	fopen()	fclose()	fwrite()	read()	close()	putchar()	ungetc()	printf()	fprint()	vfprint()
cksum	S	HC	HI						S	S2	S2
comm		HC	HC	S						S2	S2
csplir		HC	HC	S	HC	HC				S	
cut		HC	HI				S	S		S2	S2
expand		HC	HC				S			S2	S2
fmt		HC	HI				S			S2	
fold		HC	HC	S						S2	
head		HC	HC	HC	HI	HI				S2	S2
join		HC	HI	S			S			S2	
md5sum	S	HC	HC				S		S	S2	S2
nl		HC	HC	S					S	S2	S2
paste		HC	HC	S						S2	S2
pr		HC	HI				S			S/S2	S2
split		HC			HI	HI				S2	S2
sum		HC	HC				S			S/S2	S2
tr		HC		HC	HC					S2	S2
tsort		HC	HC						S	S2	S2
unexpand		HC	HC				S			S2	S2
uniq		HC	HC	S						S2	S2

Table 10-2 Gnu Text Utilities

Bin Utilities	fwrite()	fclose()	fopen()	putchar()	rename()	fprint()	vfprint()	fputs()	fgets()	fflush()
ar	S	HI	S/HI							
nm	S			S						
objdump	S			S						
ranlib					HC	S	S			
size	S			S						
strings				S				S		
strip					HC	S	S			
addr2line	S								S	S

Table 10-3 GNU Bin Utilities

Silent failure (S) occurs when an application does not crash or acknowledge any error, even though an error value was returned by the function. The application continues its execution as if no error occurred.

The final error category is also a silent failure (S2), but the function in question was used to produce an error message because another function has failed. The primary error is being acknowledged, but the function used to produce the error message itself is not checked for errors.

The test results are summarized in Tables 10-2 and 10-3. The primary observations are that the most common case is silent failures. The silent failures do not always result in no output, but very often corrupted data. Additionally, output is less likely to be checked for an error condition than input.

In a silent failure, a function call would fail to read to or write from a file and either report nothing at all or report false data. For example when failing fread in the cksum utility, the output is 0 for the number of bytes in the file and an incorrect checksum is produced as well. No error was reported even though the correct error condition was returned.

Csplit is a utility that splits a file into two separate files. When fwrite is failed in csplit, two files are created and the csplit even displays what the correct sizes should be, but the files themselves are empty. No error is given and the application exits normally.

If an fclose fails, an error is almost always given. However, the program does not always exit gracefully nor does it give clear information about what error occurred or where it occurred. For example, in the join utility when fclose() fails the error given is: "./join_paris: k: ,ðÿ¿,ðÿ¿."

FlakyPalm

While the GNU textutils were applications in use for years, some people were critical of our use of those applications. They were not developed to handle exceptional conditions. Desktop utility programs are written assuming a local data storage device. Any failure of the disk would have greater impact to the overall operating system with catastrophic effects not limited to the application in question.

Our next candidate was wireless handhelds. It was a platform to which failures, especially transient failures, would be more common. We used the Palm device emulator as our basis. It was easier to modify the code for the Palm device emulator than to devise a scheme to insert our exception injection engine into an actual handheld device. The other advantage to using the emulator is that we can verify that it is a correctly running system.

We tested seven email clients for the Palm device. We are not associating them with their specific test instances. We tested, in alphabetical order, Iambic Mail, MsgAgent, MultiMail, Palm Mail, PapiMail, ProxiMail, and TG Postman.

We tested each mail client using a POP mail server. A connection to the POP mail server was made, and one email message was downloaded from the server to the client. The email client was configured to leave the message on the server.

Palm Email Network Error Handling							
Client	**Open**	**Close**	**Connect**	**Send**	**Receive**	**GetHost**	**Select**
A	HC	S	HI	HI	HI	HI	HC
B	L	S	HC	HC	HC	HC	HC
C	L	S	L	HC	HC	HC	HC
D	HC	S	HC	HI	HI	HI	HC
E	HI	S	HI	HI	HI	HC	HC
F	HI	S	HC	L	L	HC	L
G	HI	S	HC	L	L	HC	L

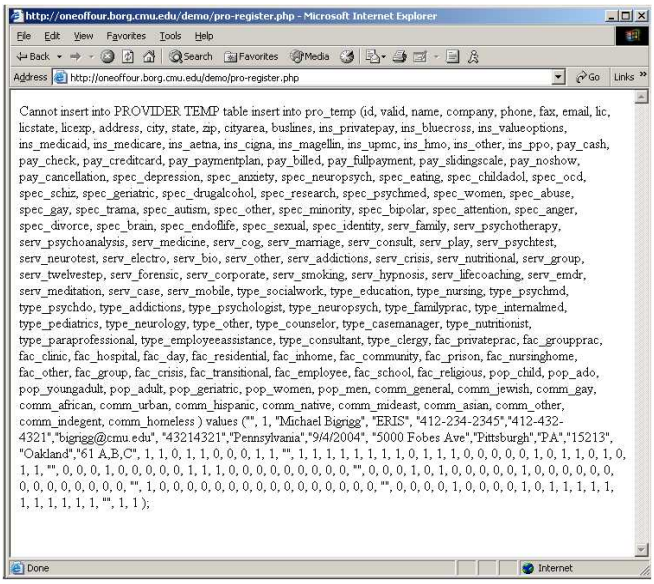
Table 10-4. FlakyPalm Results

The results, in Table 10-4, showed that wireless handheld applications were no better at exception handling than desktop utility applications. We categorized the results in a similar way as our FlakyPOSIX results for when the applications handled the exception correctly (HC), or handled it incorrectly (HI). However, unlike FlakyPOSIX, which had many silent failures (S), the FlakyPalm system experienced applications that would become stuck in an infinite loop (L). On a Palm device when an application gets stuck in an infinite loop it is necessary to kill the application by performing a soft reboot of the device.

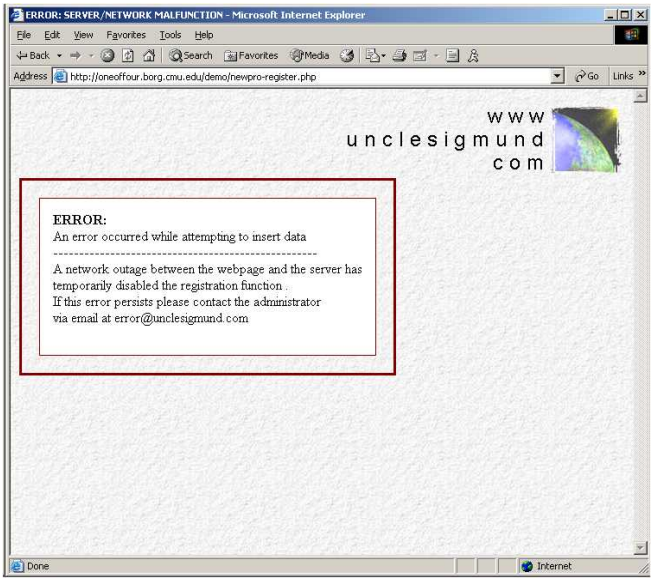
The close on a Palm device is unlike the close under POSIX. The close operation can return an exception, but it is designed to forcefully close the connection. All client applications used the forceful close, which technically exempted them from handling the exception.

FlakyNet

In an attempt to build an exception injection system using COTS parts, we created the FlakyNet system. It would sit between the client and the server as a proxy for a web server. The original FlakyPOSIX system could mimic a transient failure, but we wanted to create one without having access to the source code. In



Original Error Handling



Improved Error Handling

Figure 10-9. FlakyNET fault injection tool error handling.

addition, we wanted to transparently interpose a generic framework piece that could sit between two machines. Thus, the goal of the FlakyNet project is to provide tools that will aid in the creation of reliable distributed systems. It is a software fault injection tool that emulates network faults between the client and the server in order to identify cases where software does not properly handle network failures and to determine the effects of the failures on distributed applications and includes a network message semantic reporter for several interesting applications. The information gathered can then be used to make more robust programs able to handle network errors gracefully.

While our goals were a simple system that could be general enough to fit any application, as we progressed along the path, we inadvertently transformed our exception injection concept into a fault injection reality. The HTTP protocol did not allow for an exception to be passed from server to client and we were forced to perturbate the network connection to cause an exception to be raised. The exception given was one based on the disconnection of a network.

We ended up developing a fault injection system and not an exception injection system. The engine itself raises exceptions and does not just cause events to occur that would result in an error. This led us to an understanding that any system we build with exception injection must have explicit exceptions installed, either as part of the language, or part of the application protocol.

Figure 10-9 shows an example of the improvements that can be made after seeing the results of the FlakyNet fault injection tool.

Creation of the network fault injection tool

The FlakyNet system comprises two components. The first is a network software fault injection tool in the form of a proxy manager. The proxy will (based on user-defined parameters) emulate transient or catastrophic network failures. The second is a network message analysis tool that will aid the developer in determination of when and where the fault was injected into the network so as to aid in the identification of the problem within the application code.

In a nutshell, FlakyNet is a system for controllable fault injection analysis. It is a general framework for how/when to do software fault injection (SWFI) of storage systems, networks, etc. but in a controllable way so as to provide insight into how the software behaved. Many SWFI systems will do bit flipping or random noise injection to determine how the software will react. The problem with those techniques is that while it may show that the software might have a problem when faced with catastrophic failures or corruptions, it doesn't provide much insight into what really went wrong and especially doesn't say what to do to fix the problem.

We've applied our framework to I/O intensive programs, network-based applications, and wireless systems. The result of application analysis is that programmers will do error checking for an open, but then neglect any subsequent error checking.

We've also applied our framework to systems such as mySQL (Section10.4.1) and also to an implementation of NFS. With mySQL when faced with an error (no matter what kind or where), it exits with a bye. NFS, on the other hand, will block.

The FlakyNet fault injection tool was written in Java 1.3.1_04 and contains the classes necessary for running the FlakyNET fault injection tool in Windows 2000. The tool is available for download at http://www.ices.cmu.edu/flaky/flakynet/FlakyNet.zip

FlakyDisk

We implemented an exception-injection SCSI disk emulator, called FlakyDisk, to understand the response of the operating system I/O subsystems to faults. We converted a SCSI target mode device system available in FreeBSD 4.8. The base system uses a desktop computer running in SCSI target mode to act as a SCSI disk. An initiator machine is attached to the same SCSI bus, and communicates with the target machine as if it were a SCSI disk. The target machine emulates a SCSI hard disk. It is seen as a hard disk from initiator machine's view. The running OS is FreeBSD 4.8 Release. It contains a SCSI card that supports target mode. We used and Adaptec AHC. The SCSI commands are captured within the operating system and are then passed to a user-level application for processing.

The user-level application on the target machine responsible for the SCSI command processing was modified to insert the flaky engine, which placed its location on the system side. The boundary was drawn between the client and server processing. The SCSI protocol provides exception support.

We tested Solaris 7, Solaris 8, and FreeBSD 4.8. In all situations, we initially formatted our FlakyDisk, and populated it with files without injecting any exceptions. We then emulated a transient and then a permanent failure. Most configurations resulted in the operating systems handling correctly (HC) the error. Solaris 7 permanent exception injection resulted in an infinite loop (L).

First, we ran the FlakyDisk emulating a transient failure. We allowed five read operations to complete successfully, and then reported a failure back to the initiating machine on the sixth read request. All subsequent read requests were reported as successful. All systems (Solaris 7, Solaris 8, and FreeBSD) retransmitted the read request after the initial failure. Next, we ran the FlakyDisk emulating a catastrophic failure in which the first five read commands were reported as having completed successfully, and then all subsequent read operations were reported back to the initiator machine as having failed.

Storage Device Communication

Computing devices, in particular storage devices are no longer passive entities. Technology trends are increasing the computing power that can be cost-effectively collocated with embedded computing components, such as disk drives, graphics and network interface cards. The benefit of embedded processors for bringing products to market faster using embedded processors is well known.

These devices communicate with the host via a network using a communication protocol. A good example is a SCSI (small computer system interface) disk. A SCSI disk is a device that combines storage with

a processor. SCSI is also the communication protocol between a host computer and a storage device. Communication between the host and these devices acts as a client-server computing environment.

The processor on the device performs activities such as disk scheduling and buffer management. In addition, the processors are used to gather information on the state of the disk, making the device somewhat self-aware. Information such as the nature of error in the form of exceptions is given from the device to the host computer.

SCSI Exception Injection

A robust system is a chicken and egg problem. In order to create a system that is responsive to exceptions from underlying devices, the majority of devices must convey the exceptional conditions. In order for the devices to be built to convey their exceptional conditions, the operating system must be built to acknowledge and use the exceptions. Given a reliability-aware device, a disk, we explore the capabilities of operating systems to handle the exceptions in an appropriate manner. Anecdotal evidence has suggested that some operating systems will hang applications when the SCSI disk is not available. The question we tried to answer was: What conditions lead to the operating system hanging?

Exceptions are different than faults. A SCSI disk fault injection system would emulate such faults as bits flipped in the messages during transmission, lost messages, or the device stopped responding [Bigrigg01, Brown00]. Systems must be resilient first to faults, and then given the correct request-reply messages, the system must also be resilient to exceptions.

An exception is an informational message from a sub-system or underlying device that conveys additional information. Typically an exception is a designation of failure, but often provides more information about the cause of the exception.

For us to test its exception handling ability, the network communication protocol must support exceptions. SCSI embeds exceptions into its protocol as a characteristic of a reply message.

FlakyDisk Implementation

FlakyDisk is an exception-injection tool that tests the operating system's ability to acknowledge and respond to exceptions generated from a storage device via the SCSI protocol. It can emulate transient or permanent failures. We insert code, similar to an interceptor [Alexandrov97, Bigrigg02, Broadwell02], between the operating system and the SCSI disk. We could put the interceptor on either side of the request/reply stream between the operating system and the disk shown in Figure 10-10. In our implementation, we insert code on the side of the SCSI disk.

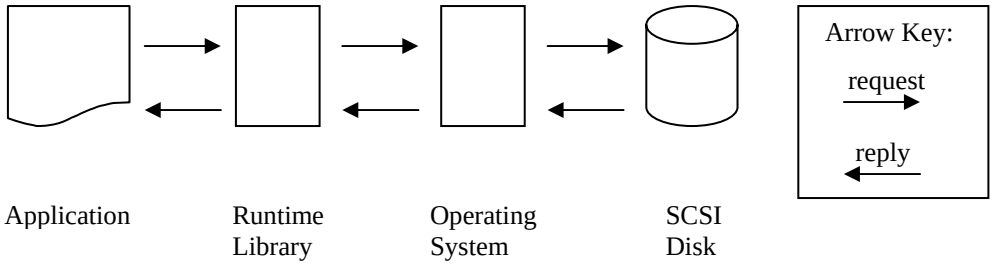


Figure 10-10. Request/Reply Stream

The request/reply between the application and the runtime system is bound by the language implementation. In the C I/O library fread call, the reply is transmitted via the return value of the function call. It will return up to but no more than the number of bytes that have been requested [Chakrabarti02]. A return value of 0 does not signify an error, but that no data is currently available such as at the end of a file. It is a negative return value that will signify an error/exception. When a disk fails or raises an exception, other

than some measure of additional reliability support, the exception should be propagated up to the application to be handled.

We need a working system that will allow us to put in our software module to intercept and modify the messages. Rather than modify an actual SCSI disk, we used a SCSI emulation system. In our implementation of a FlakyDisk, we converted a SCSI target mode device system available in FreeBSD 4.8, which emulates a SCSI disk using a standalone computer.

A SCSI card in target mode makes the machine it is on act as a SCSI client. The emulation system uses a desktop computer running in SCSI target mode to act as a SCSI disk. A host machine is attached to the same SCSI bus, communicates with the target machine as if it were a SCSI disk, and runs the filesystem to test. It uses the hard disk emulated by the target machine as second hard disk. Any SCSI cards can be used to connect to the target machine and it can run any operating system that has SCSI drivers. The set-up is shown in Figure 10-11.

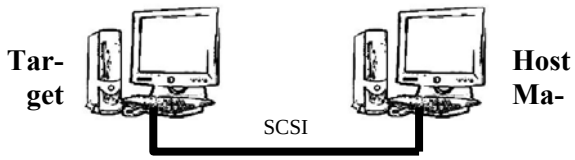


Figure 10-11. The Emulation System Configuration

The target machine emulates a SCSI hard disk using a SCSI card that support target mode. We used an Adaptec AHC. The target machine operating system is FreeBSD 4.8 Release. The target machine is then seen as a hard disk from the host machine's view.

Our exception injection system is composed on three parts: the engine, the script processor, and the identification of behavior. The engine will physically insert the exception into the reply based on the processing of a user-supplied script. Informational messages are displayed to allow the user to determine if the behavior is correct.

A. Exception Injection Engine

The exception injection engine is a software component that is added into a working system to intercept requests and either hands them off to the real working system for correct processing, or to generate an exception to be passed to the caller in lieu of a reply.

For instance, the operating system may ask to read a sector of data. The working system returns the data as payload in a reply message. Our FlakyDisk could, alternatively, construct a reply message containing a real exception to be passed back to the host as a reply message. The SCSI protocol encodes exceptions into the reply message. To insert an exception, we modify the reply message.

The host machine runs the file system in the configuration that we wish to test. We used the hard disk emulated by the target machine as a second hard disk on its own SCSI bus. The primary hard disk containing the operating system was a non-SCSI drive so as not to interfere with the SCSI messages being delivered by the target machine.

The emulation system has the SCSI messages captured within the operating system of the target machine and then passed to a user-level application for processing. We modified the user-level application on the target machine to insert our flaky engine that will selectively return a failure to the host machine. Typically the command is processed normally by the emulation system and is reported as completing successfully. It is also the case that we guarantee that the command does complete successfully.

We initiate the user-level application, which does the SCSI device emulation, by calling our modified scsi_target program to emulate a disk on SCSI bus 0, with SCSI ID of 3. The test_file is a file on the target machine to be used as local storage.

./scsi_target –d 0:3:0 test_file

B. The Script Processor

The FlakyDisk system is customizable to allow the user to determine when an exception should be raised and for how long should the exception be continued to be raised. This is not a time-based measurement but instead based on the number of command iterations that have been processed. It models the type of behavior we want to test, namely, the response of the operating system to device exceptions. For instance, we can identify that the third read request only should return an exception to model a transient error.

Our FlakyDisk system reads a file, *scsierror.txt*, which lists the exceptions that the user wants to inject. Each line in the file describes an error and has the following syntax:

SCSI_CMD ITER KEY CODE EXT

SCSI_CMD: This is the name of the SCSI command the user wants to generate an exception for. The valid commands are:

READ_10	WRITE_10
READ_6	WRITE_6
INQUIRY	REQUEST_SENSE
READ_CAPACITY	TEST_UNIT_READY
START_STOP_UNIT	SYNCHRONIZE_CACHE
MODE_SENSE_6	MODE_SELECT_6

ITER: An error will be generated when the FlakyDisk program receives the (iteration_number)th command. There are three different forms for this parameter:

n : An error is generated at the nth iteration of the command.
n-m : An error is generated for each iteration between the nth and the mth iteration.
n+ : An error is generated for any iteration greater than or equal to n.

KEY, CODE, EXT: The sense key, sense code, and the ext sense code are parameters to return in the error report. They specify the type of error that was encountered. You can find a list of the possible values that these parameters can take at http://www.arkeia.com/resources/scsi_rsc.html.

You can use the same SCSI_CMD several times if you want to generate several errors at different times for the same type of command. Figure 10-12 shows an example of the 10th READ_6 command and the 5th WRITE_6 command should raise exceptions.

```
# Syntax: SCSI_CMD IT KEY CODE EXT
# The values are decimal
READ_6 10 3 17 00
WRITE_6 5 4 12 0
```

Figure 10-12. An example of scsierrors.txt

C. Behavior Identification

An understanding of correct behavior requires a higher-level knowledge of what the behavior should be.

The file *cmdlog.txt* logs each command receives by FlakyDisk. It indicates when the command was received and displays the input parameters. Then it indicates when the command ended. Figure 10-13 contains an example cmdlog.txt file. We log the requests coming in designated as “cmd received”. The request information includes the command, such as INQUIRY or READ(6), and the associated parameters. We give an instance number to each command. This allows the user to run the system normally and then use the log to determine when an exception should be inserted. We also log the corresponding reply designated as “response sent”. If we send an exception rather than a normal response, we identify the exception information sent.

```
INQUIRY #1 cmd received, Lun: 0
INQUIRY response sent
INQUIRY #2 cmd received, Lun: 0
INQUIRY response sent
INQUIRY #3 cmd received, Lun: 0
INQUIRY response sent
REQUEST SENSE #1 cmd received, Lun: 0
REQUEST SENSE response sent
TEST UNIT READY #1 cmd received, Lun: 0
Check Condition sent, Error code: 112, Sense Key: 6, Sense Code: 41, Ext Sense Code: 1
REQUEST SENSE #2 cmd received, Lun: 0
REQUEST SENSE response sent
READ CAP #1 cmd received, Lun: 0, Logical Block: 0
READ CAP response sent
READ CAP #2 cmd received, Lun: 0, Logical Block: 0
READ CAP response sent
READ(6) #1 cmd received, Lun: 0, Logical Block: 0, Length: 1
READ(6) response sent
READ(6) #2 cmd received, Lun: 0, Logical Block: 33, Length: 1
READ(6) response sent
WRITE(6) #1 cmd received, Lun: 0, Logical Block: 768, Length: 4
```

Figure 10-13. An example of cmdlog.txt

Experimental Evaluation

We tested Solaris 7, Solaris 8, and FreeBSD 4.8. In all situations, we initially formatted our FlakyDisk, and populated it with files without injecting any exceptions. We then emulated a transient and then a permanent failure with results in Table 10-5.

Test 1.

We ran the FlakyDisk emulating a transient failure. We allowed five read operations to complete successfully, and then reported a failure back to the initiating machine on the sixth read request. All subsequent read requests were reported as successful. All systems (Solaris 7, Solaris 8, and FreeBSD) retransmitted the read request after the initial failure. Thus, all system tested were resilient to transient exceptions.

Test 2.

This time we then ran the FlakyDisk emulating a catastrophic failure in which the first five read commands were reported as having completed successfully, and then all subsequent read operations were reported back to the initiator machine as having failed.

Solaris 7 continued to send the read request. The net result was that the application responsible for the read request was hung. This did not interfere with other applications running on the machine.

	Transient Failure	**Permanent Failure**
Solaris 7	Good	Hangs
Solaris 8	Good	Good
FreeBSD	Good	Good

Table 10-5. FlakyDisk Results

Solaris 8 and FreeBSD 4.8 did not continue to indefinitely attempt to read. They properly sent the exception up the call stack and informed the application of the exception. The FlakyDisk system can be used

when developing I/O subsystem code to ensure the timely and correct processing of exceptions generated by reliability-aware devices. Using the FlakyDisk we were able to experimentally determine the cause of application hanging that was otherwise only supported by anecdotal evidence.

10.5 FlakyIO Framework

Rather than designing a system and then implement it, we built several models of exception injection in order to understand the important pieces of the design. This included one failed attempt. We design and built four versions, FlakyPOSIX, FlakyPalm, FlakyDisk, and FlakyNet. The FlakyNet version was our failed attempt.

The protocol of communication between caller and callee must provide a mechanism for providing an exception. A good example of no exceptions is the atoi() function in C. If the string provided is not a numerical equivalent, the behavior is undetermined. It is not possible to evaluate the caller's ability to handle exceptional conditions, as the function does not express those exceptional conditions back to the caller.

The FlakyIO framework begins with a working system. We then use a software interceptor technique [Alexandrov97, Dasgupta99, Hunt99, Vo97] to create a module, which we call the exception engine, within the system that uses interception to inject exceptions into an otherwise correctly running application.

There are three primary design issues in the creation of a SWEI system. They are interrelated, as one design choice will affect the ability to accomplish the others.

Exception Boundary
Injection Engine Location
Exception Pattern Specification
Error Identification

Exception Boundary

We drew boundary marks based on function calls. This allowed us the cleanest separation from caller to callee. It is also used by other robustness testing systems [Cristian95, DeVale01]. Exceptions, aside from the machine exceptions of illegal pointer access, divide-by-zero, and overflow, are raised at a function call boundary. In object-oriented systems with operator overloading, the operations are actually function calls rather than machine instructions. The operation of A = B + C is actually implemented as A = add(B,C). What looks like a machine operation, is actually implemented in software. Application features not available in hardware are implemented via a function call inserted by the compiler. This additionally provides us with feedback that our choice of function call boundary is a good one.

The I/O calls have no side effects. They do not leave the system in a corrupt state. If an error does occur, the operation can be considered atomic. I/O operations do no half-execute anything. No matter under what conditions you wish to emulate the exception occurring, you can model it as if the function call was not ever executed.

We may have to relax our no fault model approach. The fault model provides for a way to implement the side effects of the system. In I/O exception injection, it has not been necessary to construct side effects based on the faults that cause the exceptions.

Injection Engine Location

While we tried different locations for the engine, the location is basically in the application or in the system. The application side can either have the engine mechanism inserted via program transformation of the source code or a modification of the object code. When the engine is in the system, it is introduced manually into the system used by the application. The major difference between locations is in what can

be known about the function call. In the system we can only distinguish between functions, but within applications we can distinguish between the different function call instances.

For instance, in FlakyPOSIX, which has the engine contained in the application, we were able to determine that the fread was called from three different locations, but from the system side all we would be able to tell is that the fread function was called several times. In both cases we can tell the overall number of times the function was called. Only on the program side can we tell from how many places.

Its importance is based on what you are trying to measure and find out. If you are only looking at an absolute number to gauge the robustness of the application in the context of a particular system, then the system approach is suitable. If you need to know the failure condition to fix it, then source code information is necessary. As part of the source code modification, the program needs to pass along a specific instance number to be able to backtrack to the original program to tell what particular instance of the function call failed.

Exception Pattern Specification

The pattern specification is the way a user expresses when an exception should be raised. We used a file that was read upon application start to contain the pattern of exceptions to be raised. It listed all the function call along with two numbers, which expressed when to raise the exception. The first number identified when the exception should start. The second was a flag to identify either a transient (one time only) exception or a permanent (continued) exception. For instance given the following two lines:

```
fopen   3 0
fclose  5 1
```

The fopen call will raise a one time only exception the third time fopen is called. The fclose will start generating exceptions on the fifth time it is called and continue for the duration of the application.

We had envisioned a much more complex pattern specification, but this simple model was already helping us uncover a large number of problems. Our application tests did not run more than a minute and had consistent execution paths. A user interactive application would probably need a more complex pattern generation specification. The exceptions are raised based on function call name, not based on argument information. In a larger application it would be helpful to distinguish between data streams.

This file is manually generated. It was meant for programmers testing their own code. If this were a benchmark system, the exception specification would be automatically based on a full path analysis of the program to create full coverage testing.

Error Identification

This is the manual part of the process. While there are automated mechanisms to support the other phases, this one is the most difficult. A catastrophic error is easy to identify if the application/machine crashes. I/O failures do not typically cause a catastrophic failure. Incorrect exception handling behavior will consist of ignoring the exception when it should not be ignored, or trying and never giving up and returning control back to the application.

An exception identifies if the result of the operation is successful. It cannot be assumed that the result is valid unless the exception is acknowledged and determined to not identify an error. An exception must be examined.

Correct handling of an error may be to ignore the results of the operation. For instance, the failure of background saving the document as I type this paper can be ignored for some number of attempts.

There may be other alternative solutions. The system can write to a local disk rather than a remote one, should writing to the remote disk raise an exception. If the problem was a transient error, retrying the op-

eration may result in success. Care must be taken when using alternative solutions. The user may have no recourse but to kill the application if the system continues never ending attempts at alternative solutions.

We have classified the response to exceptions as HC, HI, S, S2, and L.

HC The application handles the exception correctly. This is primarily classified as behavior that will potentially try alternative mechanisms, but will ultimately return the error to the application, which in turn will report the failure back to the user.

HI When an exception is handled incorrectly, the most common behavior is reporting back to the user an incorrect assessment of the situation. This is less severe as it does acknowledge an error, but does not provide an overall correct solution.

S Silent failures are one of the worst forms of mishandling an exception. No acknowledgement of the error is given and processing continues as if the operation was successful.

S2 Many applications will use an error reporting mechanism, but do not check to make sure the error reporting mechanism does not raise an exceptional condition. We have partitioned this type of error into a sister category to the silent failures. An effective solution to an exception being raised during error reporting would be to exit the application with an appropriate return value sent to the operating system.

L Some applications will continue to try alternative solutions effectively causing the system to sit in an infinite loop.

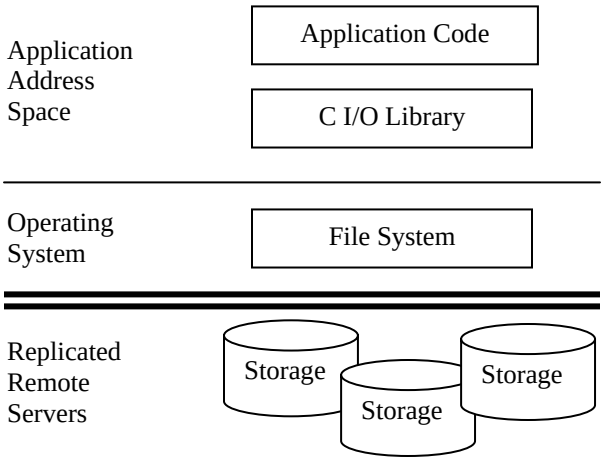


Figure 10-14: Replicated File Servers.

We did not receive any catastrophic responses. In I/O, a buffer has been allocated and it would only be the processing based on the garbage data in the buffer that would potentially lead to a crash. This effect has been explored in the Fuzz [Miller90] system. We did not implement any system that had C++ or Java runtime exceptions which, when propagated unhandled up the call stack would terminate the application.

10.6 Robustness Hinting

Many failure situations exist for distributed file systems such as network partitioning, transient connectivity due to an overloaded server, or a denial of service attack. Yet from the application's point of view all failure cases are indistinguishable from one another and simply look as if the storage server is disconnected from the network. Survivable storage systems make additional replicas of the data to be placed on different servers reducing the chance that data is completely unavailable. The redundant part of the system is the remote storage server as shown in Figure 10-14.

A survivable storage system will take a file and replicate it into pieces then disperse the replicated blocks to multiple storage servers as shown in Figure 10-15. By placing the pieces on different storage servers it is possible to improve the availability of data by increasing the chance that enough storage servers are available to recover the data. The survivable storage systems use network attached storage servers such as Network File System (NFS) or Common Internet File System (CIFS) file servers as storage servers. These systems are different than those that make use of a hardware system to implement redundancy such as disk array (RAID) systems. The RAID systems use additional hardware that is specifically allocated to provide additional data availability while many survivable storage system implementations use the resources of the client machines.

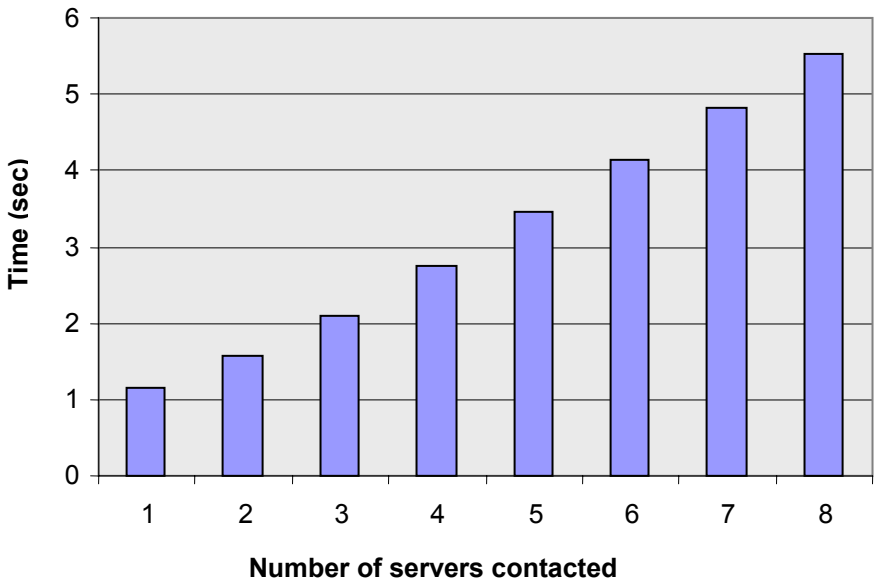


Figure 10-16: Time to complete request.

The means by which the file is replicated into pieces varies by project. A comprehensive overview of schemes used to replicate data for survivable storage systems is available here [Hunt99]. One mechanism by which a file can be decimated into many pieces is by simple replication. An exact duplicate of the file is created and each copy is distributed to a different storage server. Many survivable storage systems do not disperse data for pure reliability concerns but also for security. Each replica may be encrypted before it is transferred to a remote storage server. Additional systems will replicate data into shadow copies such that a subset of replicas is required in order to reconstruct the original data rather than a single copy. There are also additional hybrid schemes that combine replication, encryption, and shadow copies.

In all cases there are more replica blocks than the original block so it is possible to survive in spite of some number of storage servers that are unavailable, unreachable, or overloaded. At the same time as blocks are broken up into replicas it is the case that only a subset of the total number of replicas are needed and thus only a subset of storage servers are needed in order to recover the data.

We have implemented a prototype system to retrieve data simultaneously from multiple remote sources in order to test the impact of the I/O resource requirements of a survivable storage system. The system uses the NFS protocol to communicate between client and servers. Our prototype system is a user-level appli-

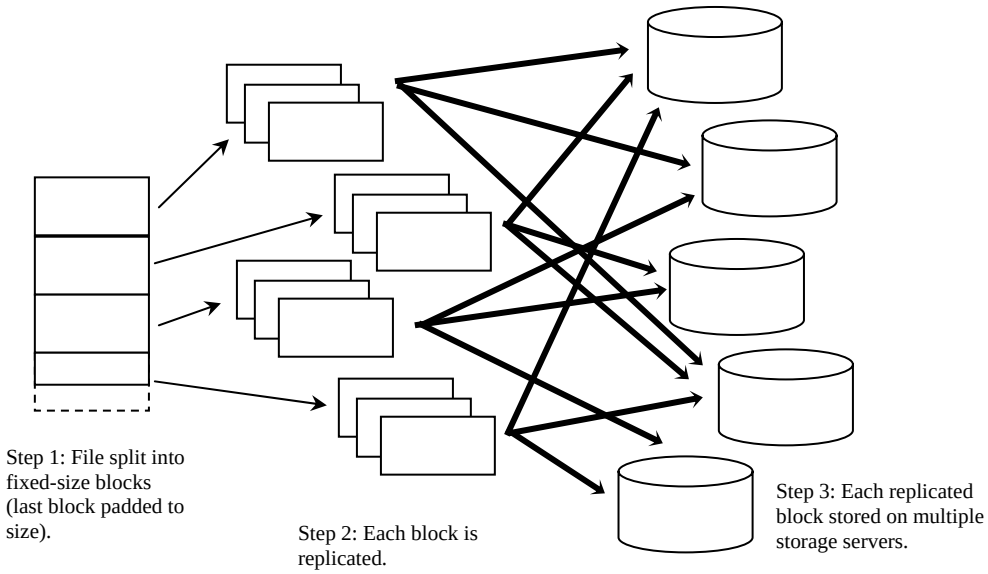


Figure 10-15: File Replication and Dispersal.

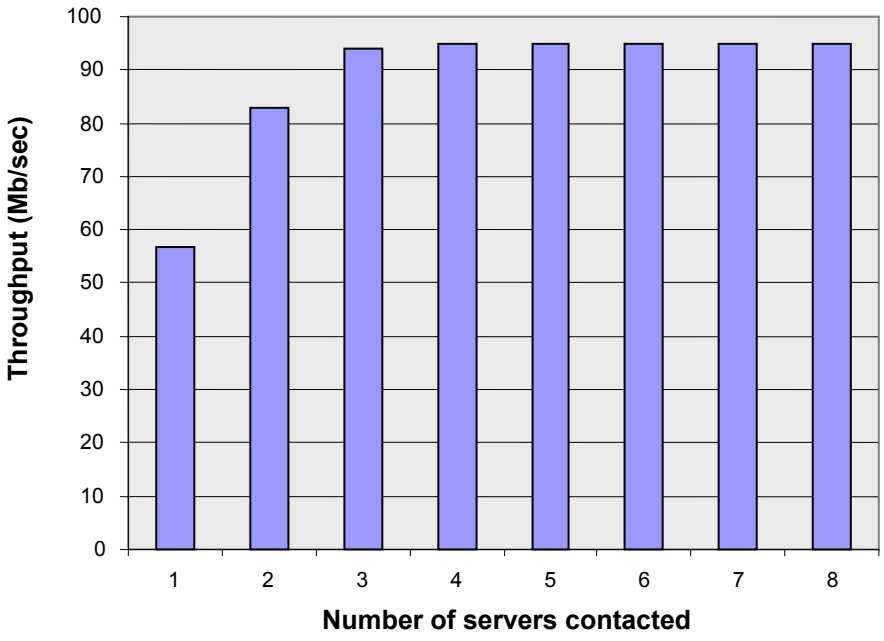


Figure 10-17: Bandwidth usage.

cation that implements the NFS remote procedure calls to mount a remote drive, create a file, write a single 8KB block of data, and then read that block of data 1000 times, thus reading an entire 8MB file. Testing and measurements were taken only during the read phase.

The testbed consisted of nine PC systems interconnected via a dedicated 100 Mbps switch. Each system contains a 600 MHz Intel Pentium III, 256 MB of RAM, a 3Com Fast Etherlink XL network interface card, and a Quantum Atlas 10K disk connected via an Adaptec ULTRA-2 SCSI controller. One system, acting as the client, runs SuSE Linux. The other eight systems, acting as NFS storage nodes, run a standard NFS server on SuSE Linux. Time measurements are based on the Pentium cycle counter, and all values are the mean of 60 measurements.

We simulated a survivable storage system that retrieves information from multiple sources. We varied the number of remote storage servers from one to eight servers. Figure 10-16 shows the total time required for each run to complete. Not surprisingly, the cost of contacting N+1 servers is greater than that of contacting N servers. When contacting three or more servers simultaneously we were able to use all bandwidth that was available to our client as shown in Figure 10-17. This greatly impacts the time it takes to complete the request. Without any understanding of the needs of the application, the assumption would have to be that the file system request data from all storage servers that have replicas reducing the response time for all applications. Otherwise under heavy load the file system could request one replica for each application that can handle less than full availability of data and all replicas for those that have a greater robustness need. This would reduce the overall penalty to all applications.

Consider two applications, A and B, each reading a four-way replicated 8MB file. The worst-case situation is that the file system makes simultaneous requests to eight storage servers for data, all of which are up and active. As data arrives and saturates our network connection, the servers compete for bandwidth causes an overall delay in satisfying the request. To request data from less than all servers could lead to the situation that we are unable to satisfy the requests as our time-out has expired and we must return control to the application.

This would be the situation should we be unaware of the needs of the application. However, if we assume that A does correct error handling and B does not, we would want to optimize the number of servers contacted for application B. We want to be able to work harder for B than in order to satisfy the read requests before we have to time out and return a failure to application B, which would produce an incorrect result should it receive less than all of the data it has requested.

Knowing that B has a greater demand for data availability we could choose an alternative to contacting all eight machines at once weighting the number of servers for B. If we contacted only a subset of servers,

one server for A and then all servers for B, the responses returned in 62% of the time of using eight servers.

Application Error Handling

To understand the disconnection between applications and the underlying system, the path of control from the application to the storage will be shown. As control passes between components as shown in Figure 10-18, only limited control information is passed from the application to the lower layers.

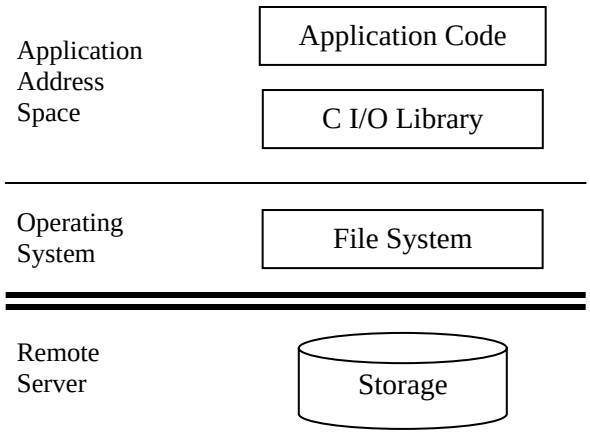


Figure 10-18: Component Boundaries.

A distributed file system is an operating system interface to the user or application that presents a remote server as if it were a local storage device. The distributed nature of the file system is hidden from the application and is presented in the same way as a local file system. While this abstraction permits faster software development, software development seems to assume only the local file system characteristics and its unlikelihood of storage failure.

Applications use libraries linked in with the application to interface to the file system just as it would for any local storage device. For instance, the interface that C applications have to the file system is through the C standard I/O library.

The information provided from the application to the C I/O library fread function is (1) a handle to the file to be read which identifies which file and where the last read ended, (2) the amount of data to be read, and (3) a location of where the read data should be deposited within the address space of the application. No control information that specifies the needs of the application is passed.

The library, which may buffer data that has not been delivered to the application, may have sufficient data to complete the request. Otherwise, the library will in turn make a call to the file system for data, often requesting more than is necessary in order to minimize overhead in crossing the boundary between the application and the operating systems. The file system may contain the data within its cache and could fulfill the request, as there is an even greater overhead in crossing machine boundaries. If the file system does not have enough data, it will request data from the remote file server. After receiving data a portion is transferred to the library that in turn transfers a portion of the data to the application as requested.

In addition to the data being passed, status information is also returned from the system to the application during the fread call. This status information overloads the return value to signify either the number of bytes read, an indication that file is at the end, or that an error has occurred.

An error would be returned back to the application through the library should the remote file server be unavailable. It is up to the calling application to check the return value to determine if an error has occurred or if the file is at its end.

If the file server is unavailable and the application does not check its return value, the application will process whatever garbage data was in the area it set aside for the fread result.

This rest of this section summarizes our previous work in the analysis of application error handling with a faulty storage system. Additional details are available [Alexandrov97].

We built a test harness to allow us to selectively manipulate the behavior of the I/O function calls of a program to simulate storage system failures using fault injection. We applied the testing to a specific class of applications that are I/O based, use the C standard library, and have been around for years, the GNU

binutils and textutils. Our testing harness performed source code modification to insert a fault injection layer into the application. Faults were injected into a single execution of the application based on the specific function call. To fail the fread call, a –1 was passed as the return value to the calling application.

We ran the applications on a failure free environment and then again inserting faults and observed their behavior. The results fell into four groups: handles correctly (HC), handles incorrectly (HI), silent failure (SF), and silent error (SE). Many applications handled error conditions correctly, the HC group. All correctly handled the failure of an fopen call. Some applications acknowledged that an error did occur but misreported the error, the HI group. This was due to the engineering of the testing system that emulated transient failures instead of catastrophic failures. A transient failure would only fail for a specific instance of a function call but would subsequently not fail for the next instance of an I/O function call. An application acknowledged a fread failure but used a feof function call to determine if the file is at its end. The read call will fail but the feof call would not. An additional coupling of function calls is for silent errors, the SE group. Most applications used a print in order to report an error and did not check to see if the error had been correctly reported. While it may seem as if printing an error message is the only recourse, the applications in the SE group did not return an error code to the operating system to report that the application had failed.

The most troubling were the silent failures, the SF group. There were two interesting causes of silent failures. The first is a group of applications that do not check the return value of a function call. The second group had a less subtle reason was due to the fact that many return values were overloaded to provide much information. The return value from a read call can be a positive number (the number of bytes successfully read), a zero (there is no data currently available such as an end of file), or a negative number (an error). Applications checked the return value to determine if it was greater than zero, assuming that otherwise the file is at its end ignoring the possibility of a negative value.

In addition, we analyzed by hand one program to determine if the lack of error checking was consistent through the entire application. The wc program is part of the GNU textutils collection of programs. Its purpose is to count the number of lines, words, and characters in a file or files identified on the command line. The v2.0 program consists of 371 lines or 118 lines of code. It can be identified to have been in use for the past 16 years (from 1985 to 2001). It was written in C and consists of four functions. There were no instances of control flow issues where error checking only exists in a subset of execution paths. A hand analysis of the main source program was performed to detect a failure to check for an error condition. The results are summarized in Table 10-6.

Routine	**Total**	**Checked**	**Unchecked**
fprintf	1	0	1
Printf	7	0	7
Puts	1	0	2
putchar	1	0	2
fstat	1	1	0
lseek	2	2	0
read	3	3	0
open	1	1	0
close	2	2	0
setlocale	1	0	1
SUMMARY	**20**	**9**	**11**

Table 10-6: Hand analysis of the wc program.

A simple calculation of the number of checks that should be performed against the number of checks actually performed results in a 45% reliability rating. The usefulness of this rating is not promoted as it does not reflect the frequency of each call, but can be included as a guide to understand the program behavior.

The major assumption that was made by the wc program is that all output is guaranteed to succeed. It is possible for the system to take additional measures in order to preserve that assumption. While there may be a large cost associated with establishing that guarantee, the system does not have to take additional measures for other requests such as file manipulation or input as the application can handle those failures.

There have been many additional approaches used for program analysis especially in the detection of software faults. These systems are typically aimed at providing information to the programmer in order to modify the program source to eliminate software faults.

One group of fault analysis techniques similar to our own will run the entire program or a subset of the program to observe its behavior. Errors are often introduced to examine how the software behaves. This approach makes use of external faults being passed into the application, which will cause it to fail. Such approaches include: fault injection through random memory corruption, passing values typically known to cause exceptions into an individual software module through its software interface [Cristian95], or the creation and use of a comprehensive test suite. In particular it is possible to identify the type of input or condition that has led to the fault, but it does not identify any remedial action that should be taken by the application.

Compile-time analysis attempts to identify program features that would cause a program to behave improperly. The analysis focuses on a particular characteristic that is typically the base cause of faults such as portability [Dasgupta99]. Other approaches use programmer-defined models of acceptable behavior to drive the analysis [Bigrigg02, Broadwell02]. These systems report the results to the programmer for correction presupposing that the programmer has the time and capability of rewriting the offending code to correct it.

Implementing Robustness Hinting

We propose an automated solution to fixing the errant code. There are many ways to couple program information with the underlying system. A simple approach could be the use of one or more global variables such as is used in the errno method of C [Chakrabarti02], but instead of passing a characteristic from the subsystem to the application we would pass the characteristic from the application to the subsystem. Inserting function calls within an application is another mechanism that could be used to transmit program characteristics between the application and the subsystem. This is the approach that is used in the tainted mode in Perl [Engler00]. Data that is has been received from outside of the application is considered to be tainted and therefore should be treated as suspect. Again, the characteristic is provided to the application from the subsystem.

The use of a global variable to transmit additional control information is simple and straightforward but it is not thread-safe. Making a function call prior to an I/O function call to transmit the information is an alternative solution, but is as failure prone as the function calls we are trying to fix and another additional tool would have to be created in order to make sure that for every I/O call there is a robustness hint call.

Our approach is instead to replace the C I/O library with the PASIS I/O library. The signature of the function calls is modified to include an additional parameter in its function calls to transmit the hinting information. This allows us to use the compiler to detect the missing robustness parameter when modifying existing code.

An alternative is to use a hinting mechanism similar to what has been done for file prefetching [DeVale01] would pass information from the application itself to the file system. A robustness hint would be passed for each I/O function call that had the correct error handling code. While optimally automated hinting is desired, preliminary hinting would use the manual insertion of hints into the application code.

The argument against this is that if it is possible to know where the problem lies and the source code is available, it is possible to fix the code. However the insertion of hinting code, which only requires the programmer to acknowledge the problem, would be far easier than the modification of the program logic. In addition, the hinting is added only where there is handling of error conditions allowing the file system to default to its current behavior of blocking should no hint be given.

It is true that programmers often overlook error checking, as they believe that the various errors are just inconceivable and do not need to be checked or are overwhelmed with the task of identifying all possible error situations. Unhandled error conditions would lead to potential software failures when the underlying system cannot satisfy our requests. Programmers alone cannot thoroughly account for all error conditions, as it is a non-trivial task with a lack of tool support. But as shown in the previous section while error checking is difficult, it is not always missing.

Robustness Hinting Conclusion

Systems should only take exceptional measures to ensure robustness when robustness measures are not handled by the application. We showed:

(1) There are techniques to increase the chance that data storage is available. However, these techniques are resource intensive and have a negative impact on other applications running on the system.

(2) Some I/O applications can handle errors on their own and do not need additional support. Some applications do not handle I/O errors and will silently process garbage and produce and incorrect result.

(3) By hinting to the file system the application's ability to handle I/O errors it is possible for the file system to make better resource allocation decisions. This will reduce the impact on other applications when it needs to be avoided.

In summary, 1+2=3.

10.7 Summary

We have learned much from implementing a variety of exception injection systems. It is a useful approach to the research of designing systems. We believe future work should explore the needs of much more diverse applications.

While we developed several exception injection systems, we also want to explore non-I/O based systems. I/O systems are simpler as operations do not have side effects. Non-executing or partially executing I/O operations are atomic. To account for systems having functions with side effects, we may have to implement a fault model to be able to modify the internal state of the function based on the type of fault that caused the exception.

One place where we have not fully developed all the parameters is the exception specification, which is overly simple. While we would not want a more complex system just for the sake of having one, we need to explore the types of applications that are not served by a simple specification.

By building several exception injection systems, we were able to evaluate the design choices made. The implications and effects of our choices such as exception boundary partitioning, exception engine placement, exception pattern specification, and error identification were shown.

The focus of our exception injection was data intensive applications, wireless handheld applications, storage systems, and network applications. We presented results which show that every system has application instances that do not do correct exception handling.

11 PASIS AND JBI

This chapter looks at how PASIS technologies can be integrated and utilized to enhance the survivability of JBI (Joint Battlespace Infosphere) systems. The PASIS architecture combines proven technologies to construct information storage systems for which availability, confidentiality, and integrity policies can survive component failures and malicious attacks [Wylie00]. Specifically, PASIS-enhanced storage systems replace centralized servers (which are single points of failure for security and availability) by distributing information in a manner that is consistent with user's requirements, over a set of decentralized servers. Our primary research focus is the provision of a general framework that quantifies and mitigates the performance consequences (and trade-offs) of replacing conventional storage systems with survivable storage.

To understand how PASIS-enhanced storage can be used in JBI systems, it is instructive to consider what PASIS is and what it is not. The PASIS architecture hardens information storage, but the protection mechanism does not extend to the information processing aspects. For example, PASIS-enhanced storage may choose to protect data using secret-sharing schemes in such a way that data is divided into multiple pieces where each piece alone exposes little or no information. Only when enough pieces are combined together, does the original information reveal itself. The filters, fuselets, client displays, and/or Oracle database components may see the data, in its entirety, after the various pieces are retrieved from the PASIS storage. In practical applications, many operations (except perhaps digital signatures) need access to the original data in cleartext. Securing the data at these points is outside of the scope for PASIS; other protections will be needed. PASIS will complement such other mechanisms by safeguarding the repositories that hold data not currently in use.

The JBI project seeks to create specialized, collaborative information systems that will help military personnel more effectively utilize available data. Still in its early conceptual phase, a JBI system is expected to involve a number of different data processing and delivery components. For example, publish/subscribe components might help with collaborative interactions, while fuselets can process data streams and produce new information objects.

As with other applications that involve large-scale information dissemination, a critical subset of the information available to JBI users will come from persistent storage (other info will come from active surveillance sources). The accuracy and availability of the stored information is therefore critical at exactly those times when the JBI system will require these resources (e.g., during battlefield military operations).

The most natural and elegant way to integrate PASIS-enhanced storage into JBI is to replace and thereby harden the storage mechanisms that would otherwise be employed by the JBI system. This approach requires minimal changes to the existing architecture—it impacts only the storage while leaving the rest of JBI unaffected. This is especially beneficial considering JBI is still in its design stage; the JBI architects need not be concerned with the nuances of PASIS. Their design can evolve in a manner that is largely independent of the underlying storage mechanisms.

Our work aims to identify and design the appropriate interfaces between the PASIS storage and the JBI system, and to evaluate the performance costs and survivability benefits of integrating such storage technologies. This document describes our effort in designing and implementing a prototype PASIS-enhanced storage system to meet JBI-like storage requirements, and reports on our exploratory models and benchmarking results.

While efforts must be expended in terms of interface amalgamation, we envisage a straightforward integration process with minimum cost. We also note that, independent of the JBI platform, clients of JBI can take advantage of the PASIS technology to secure their critical data, in effect providing much stronger overall protection of persistent data.

It should be noted that another important aspect of PASIS is its configuration flexibility. By selecting from among thousands of data distribution scheme options, PASIS storage can be positioned at desired points in the security vs. availability vs. performance trade-off space. With this flexibility, JBI commanders can match their deployed system to their needs. Even more, when systems transition from one task or one commander to another, they can be reconfigured to match the situation. This trade-off space is described and explored herein, as well.

Our exploration of how PASIS technologies apply to JBI systems have produced several insights and results, including:

- PASIS storage can be cleanly integrated into JBI systems as survivable storage for JBI information repositories. Such repositories will be necessary for users interpreting the events delivered via the publish-subscribe portion of the JBI system. By "cleanly integrated", we mean that no changes are needed to the software (e.g., an Oracle database) that stores and retrieves the data. PASIS libraries can be plugged in beneath the database transparently.
- Specifically, we have demonstrated that PASIS can be cleanly integrated into a system running Oracle. This required building an NFS front-end to the PASIS libraries, which could be expected in a commercial PASIS system. It also required significant porting and tuning effort to get Oracle working properly on Linux, the system on which the PASIS-NFS prototype works; this also would not be an issue for a commercial offering of PASIS. With the base infrastructure in place, running Oracle atop PASIS-NFS was straightforward.
- By comparing Oracle running atop PASIS-NFS to Oracle running atop non-survivable NFS, we quantified performance costs associated with the PASIS techniques. To quantify these costs, we ported industry standard benchmarks for the data mining types of activity that we expect would characterize many JBI storage interactions. Numerous data are found in this report, but the most interesting finding is that the performance costs are significantly lower than originally expected. The computation and user interfacing times of the Oracle database hide many of the I/O costs introduced by the PASIS mechanisms. Obvious in retrospect, this insight makes it even more feasible to employ PASIS techniques to harden JBI storage.
- The flexibility of PASIS's survivability mechanisms is very attractive for JBI environments. Specifically, by configuring PASIS statically or dynamically, a commander can match the JBI storage's security, availability, and performance characteristics to mission needs. In dynamic settings, such as JBI, this is vastly superior to one-size-fits-all options. PASIS storage offers thousands of options in the trade-off space, and consequences of these trade-offs are discussed in the report.

This section outlines how PASIS technologies can fit into and enhance the survivability of Joint Battlespace Infosphere (JBI) systems. We first give a high-level description of a JBI system. Then, we discuss the approach for using PASIS-enhanced storage in JBI systems. Finally, we discuss a more radical idea in which PASIS can be used beyond the storage to enhance the survivability of the entire JBI system.

11.1 Joint Battlespace Infosphere

The Joint Battlespace Infosphere (JBI) system is an information management system designed to facilitate information aggregation and dissemination to users at all echelons of command. The basic JBI functions center around the following tasks:

- Fusion and distribute information
- Collect and store information

A JBI system is an information management process involving clients interconnected via a network to a set of core JBI services. The core JBI infrastructure is a publish/subscribe network, which consists of a

distributed set of information brokers. This information management system exploits the wealth of data currently fielded by non-interoperable information systems by collecting data and transforming it into a common form (object schema) in order to provide an Integrated Information Base from which a battle-space common operating picture can be generated.

Clients of JBI include publishers, subscribers, and fuselets, a class of objects whose purpose is analyzing incoming information streams and publishing new information objects. A client can subscribe or actively query information from the core JBI infrastructure. When a relevant information object is published, the JBI system delivers the object to the interested clients. Information dissemination is immediate and near real time in JBI. In addition, the JBI system also maintains information repositories, which are archived databases of published information objects. The repositories facilitate queries of information objects and offer a starting point for post-operations analysis. A high-level JBI architecture is depicted in Figure 11-1.

11.2 PASIS and JBI

The information databases in JBI can be part of the JBI core infrastructure, or they can interact with JBI via a client adapter. When the JBI brokers receive a subscription that matches the information advertised by the database adapter, the broker queries the adapter, which in turn returns the relevant information object. JBI client objects such as fuselets are likely to make extensive use of the information repository in order to transform data into knowledge.

The repository databases may contain critical JBI information objects, such as battlefield plans and enemy position information. It is conceivable that some of these repositories may operate in an untrustworthy environment. The reliability, availability and security of the repositories, therefore, are of critical importance. This is exactly what secure storage technologies such as PASIS can provide. We envisage a PASIS-enhanced storage system behind every database interface in a manner that is transparent to the end users as well as to the JBI system at large.

As an example, Figure 11-2 depicts a JBI database repository, before and after being outfitted with PASIS. In Figure 11-2a, the adapter is prompted to query the database and returns relevant information objects. The database storage is likely local to the adapter object and is only accessible through the adapter. If the storage fails, the information in the repository is not available for access to any JBI clients.

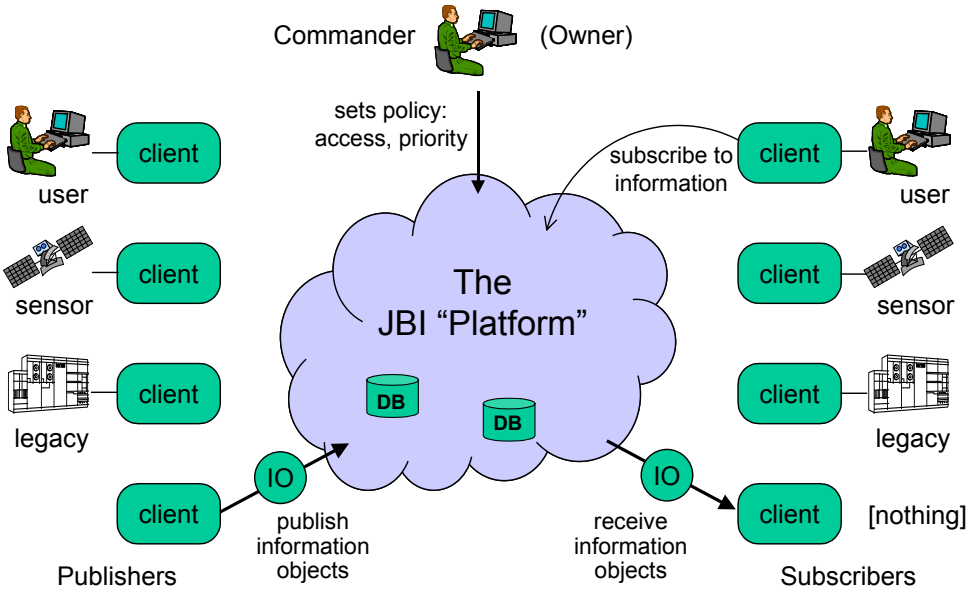
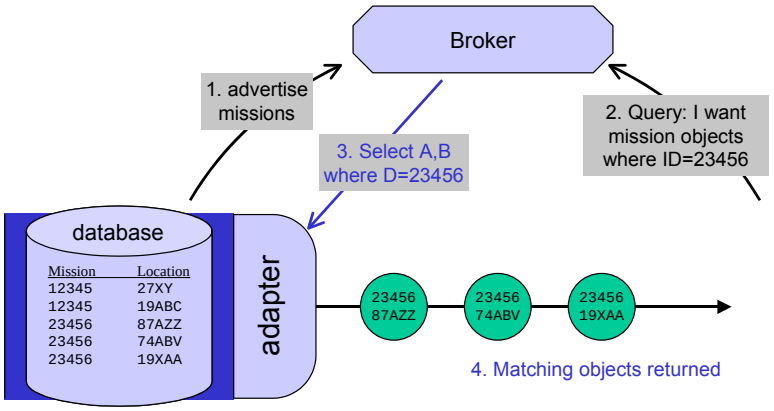
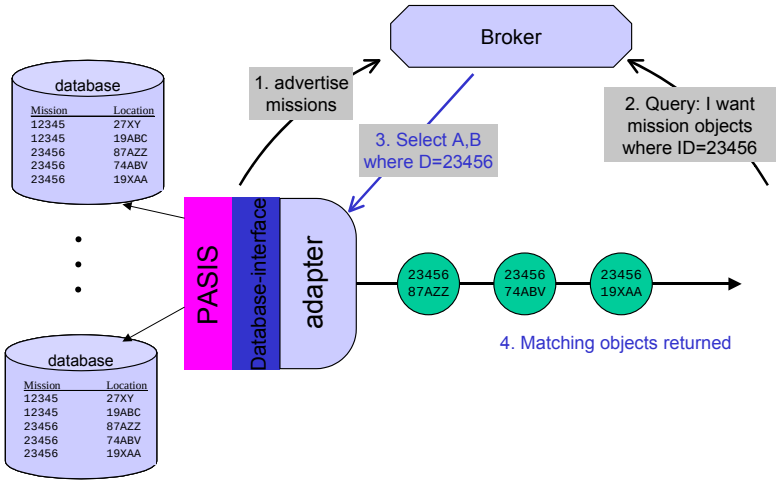


Figure 11-1. A JBI Architecture



11-2a. An Information Repository in JBI



11-2b. A PASIS-enhanced Information Repository

Figure 11-2. PASIS-enhanced storage for JBI

In Figure 11-2b, the database is no longer stored as a single copy at one location; it is replicated at various storage locations[2]. The adapter, as before, sees the same interface with the database and is oblivious of the data replication at the background. The database server (e.g., Oracle), upon receiving the query request from the adapter, will interface with the PASIS system to retrieve the relevant data. When the appropriate storage scheme is chosen, PASIS provides the mechanism to meet the performance, security and availability requirements of the user application (in this case, the JBI system).

The adoption of PASIS is transparent to the rest of the JBI system. This feature is especially attractive because a JBI system must accommodate a wide range of dynamic client needs. The less built-in functionality there is in the core JBI platform, the more extensive and flexible the system will be.

To experiment with the use of PASIS-enhanced storage in a JBI-like environment, we have built a prototype database system with PASIS as the underlying storage technology. Our implementation demonstrates the feasibility of incorporating secure storage technologies such as PASIS in real information management system and allows us to quantify its performance implications. We tested our prototype implementation using TPC-H and TPC-C benchmarks. TPC-H and TPC-C are benchmarks for decision and transac-

[2] In this example we use straight replications, but PASIS can handle secret-sharing, information dispersal and other kinds of storage schemes.

tion processing. These benchmarks embody characteristics that are typical of real world database applications. In particular, the decision support benchmark, TPC-H, generates a workload that is representative of JBI storage applications. We believe that structuring experiments with these benchmarks allows us to reason about the efficacy of PASIS in supporting practical database applications likely to arise in JBI environments. The characteristics of the benchmarks and the results of our experiments are described in Chapter 12.

11.3 A PASIS-enhanced Proxy for JBI

An alternate way to use PASIS-enhanced storage in JBI systems is to take a major departure from conventional information processing architectures. In the alternate architecture, conventional information storage would again be replaced with PASIS-enhanced storage. However, rather than putting the storage at the bottom of the architectural diagram, Figure 11-3 depicts an alternate design with survivable storage in the center of the system.

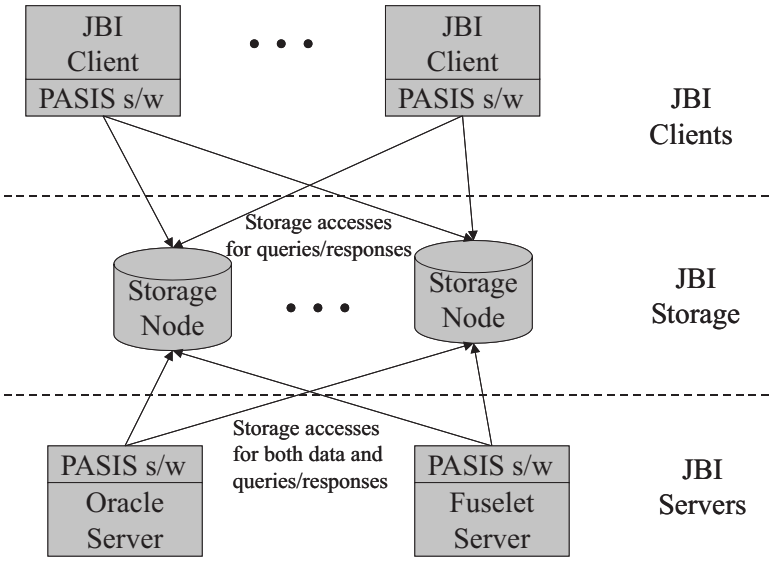


Figure 11-3. An alternate JBI architecture with PASIS-enhanced Storage

With this design, interactions between JBI clients and JBI core services occur indirectly through the one component whose survivability we know how to harden: PASIS-enhanced storage.

Abstractly, the way that this could work is that the request packets that would normally be sent from a JBI client to the JBI services are instead written to PASIS storage nodes. JBI services could then fetch these packets from storage and process them as they would normally. The same indirection would occur for data transfers in the opposite direction.

The potential positive aspect of this architecture is the ability to isolate JBI services and thereby protect them. These servers will require access to cleartext information, and therefore are a serious target for security attacks. In this alternate architecture, JBI clients may have no direct interactions with (or even knowledge of) the particular JBI servers with which they interact. In fact, it is possible to implement this system in such a way that it is physically impossible for other computer systems to communicate directly with JBI servers, which has intriguing security potential. In addition, all communications could be logged for some period of time to allow intrusion diagnosis [Strunk00].

12 DATABASE BENCHMARKS

As we have described the mechanisms of PASIS and its trade-off framework, we now turn to the application of PASIS-enhanced storage system characteristics. In JBI, for instance, a storage system would exist behind the database interface, and we used standardized benchmarks to emulate the workload the system would experience. We implemented the TPC-H benchmarking suite because it closely represents the type of queries that would be given to the database.

First, we will briefly overview the fundamental terminologies in database systems, and then go on to discuss the benchmarks.

Database Management System – DBMS: DBMS is a software system that enables users to define, create, and maintain the database and provides controlled access to this database. These systems typically offer ad hoc query facilities to many users and they are widely used in business applications.

A database management system (DBMS) can be a complex set of software programs that controls the organization, storage and retrieval of data (fields, records and files) in a database. It also controls the security and integrity of the database. The DBMS accepts requests for data from the application program and uses the underlying file system to persistently maintain the data (REF2).

The DBMS itself will maintain the integrity of the database by not allowing more than one user to update the same record at the same time. The DBMS can keep duplicate records out of the database; for example, no two customers with the same customer numbers (key fields) can be entered into the database.

From a technical standpoint, DBMSs can differ widely. The terms relational, network, flat, and hierarchical all refer to the way a DBMS organizes information internally. A database management system may provide one, two or all three methods. This does not reflect the way in which data is stored.

Requests for information from a database are made in the form of a *query*, which is a stylized question. The set of rules for constructing queries is known as a *query language*. Different DBMSs support different query languages, although there is a semi-standardized query language called *SQL (structured query language)*.

Relational Database Management System – RDBMS: RDBMS is a type of database management system (DBMS) that stores data in the form of related tables. A table is a collection of records and each record in a table contains the same fields. Certain fields may be designated as keys, which mean that searches for specific values of that field will use indexing to speed them up. Relational databases are powerful because they require few assumptions about how data is related or how it will be extracted from the database. As a result, the same database can be viewed in many different ways.

An important feature of relational systems is that a single database can be spread across several tables. This differs from flat-file databases, in which each database is self-contained in a single table. Almost all full-scale database systems are RDBMSs.

A query: A user's (or agent's) request for information, generally as a formal request to a database. It is a request for information from a database. There are three general methods for posing queries:

Choosing parameters from a menu: In this method, the database system presents a list of parameters from which the user can choose. This is perhaps the easiest way to pose a query because the menus guide the user, but it is also the least flexible.

Query by example (QBE): In this method, the system presents a blank record and lets user specify the fields and values that define the query.

Query language: Many database systems require user to make requests for information in the form of a stylized query that must be written in a special *query language*. This is the most complex method because it forces user to learn a specialized language, but it is also the most powerful.

The standard user and application program interface to a relational database is the structured query language (SQL). SQL statements are used both for interactive queries for information from a relational database and for gathering data for reports.

12.1 TPC-H Benchmark

TPC Benchmark™ H (TPC-H) is a decision support benchmark [TPCC01]. It illustrates decision support systems that examine large volumes of data, execute queries with a high degree of complexity, and give answers to critical business questions. It evaluates the performance of the illustrated system by the execution of sets of queries against a standard database under controlled conditions.

There are 22 queries designed for the TPC-H benchmark. These queries have the following characteristics:

- can answer real-world business questions,
- can simulate generated ad-hoc queries (e.g., via a point and click GUI interface),
- are capable of handling complex transactions,
- have a rich variety of operators and selectivity constraints,
- able to generate intensive loads on the database server,
- are similar to an on-line production database.

Apart from the queries, TPC-H defines two refresh functions to model the loading of new sales information (RF1) and the deleting of obsolete sales information (RF2) from the database.

The primary performance metric of the TPC-H benchmark is the Composite Query-per-Hour Performance Metric (QphH@Size). The TPC-H benchmark also reflects multiple aspects of the system's capability to process queries. These aspects include the selected database size against which the queries are executed, the query processing power, and the throughput with multiple concurrent users.

TPC-H Business Model: TPC-H benchmark models a decision support system for a business that manages, sells, or distributes products on a worldwide scale [Poess00]. The decision support system is used to compute business trends to support future business decisions.

The core of the TPC-H benchmark is comprised of a set of business queries designed to exercise system functionalities in a manner representative of complex decision support applications. These queries represent the activities of a wholesale supplier.

DATABASE SCHEMA: Figure 12-1 shows a diagram of the TPC-H database schema, which consists of eight indiviual tables. The arrows between the tables show a one-to-many relationship. For instance, a record in the Part table points to many records in the PartSupp table. This relation represents that each part has many part suppliers.

The TPC-H database schema uses the third normal form (3NF). Normalization is a process of organizing data to minimize data redundancy. It usually involves dividing a database into several tables, and the objective is to isolate data so that additions, deletions, and modifications of a field can be made in just one table and then propagated through the rest of the database via the defined relationships between tables. There are three main normal forms, with increasing levels of normalization:

- First Normal Form (1NF): Each field in a table contains different information.
- Second Normal Form (2NF)**:** No field values can be derived from another field.
- Third Normal Form (3NF)**:** No duplicate information is permitted.

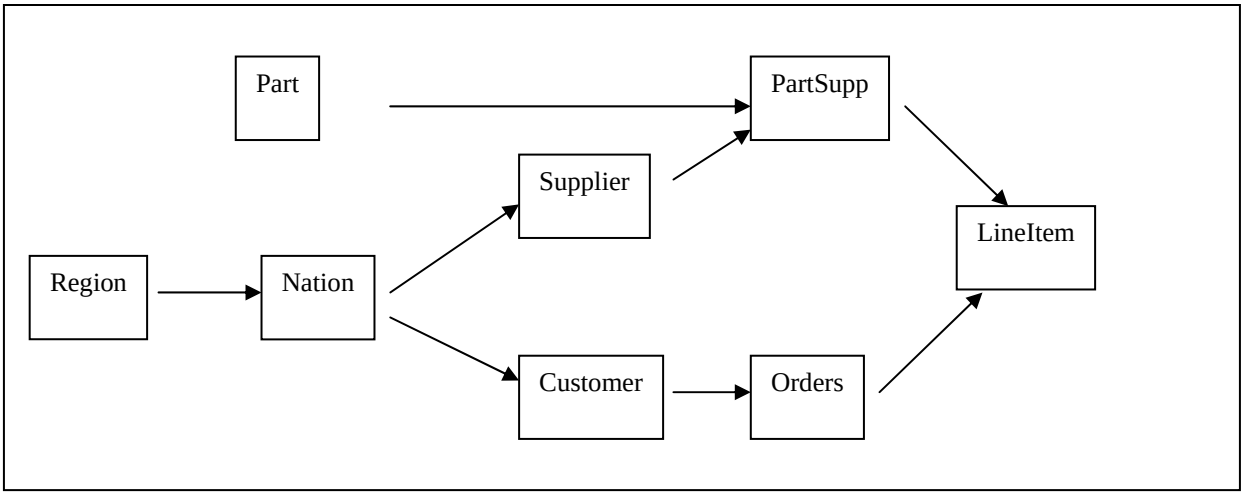


Figure 12-1. TPC-H Database Scheme

We created the schema by following the rules stated in the TPC-H specification document. The scripts used in creating the schema can be found in Appendix B.

After creating the schema, the database needs to be populated before it is ready to accept queries and refresh requests. TPC calls the population step the "load test". This step is not included in any of the performance measures. The performance test, which includes querying and refresh function runs, creates the data for the performance measures.

LOAD TEST – database load: The load test begins with the creation of the database tables and includes all activity required to bring the system to the configuration that immediately precedes the beginning of the performance test. This phase does not execute any queries.

TPC provides a data generator, *dbgen*, to generate data for all the tables. Dbgen generates its data using a Scale Factor (SF). SF determines the size of the raw data that will be inserted to the database. For instance, SF=100 means that the sum of all tables equals 100 gigabytes. Dbgen allows the creation of data with SF values of 1, 10, 30, 100, 300, 1000, 3000, and 10000. The size of all tables, except for nation and region tables, scales proportionally with the scale factor. Table 5 lists the cardinality of each database table in rows.

Table Name	Cardinality in rows
Region	5
Nation	25
Supplier	SF * 10,000
Part	SF * 200,000
PartSupp	SF * 800,000
Customer	SF * 150,000
Orders	SF * 1,500,000
LineItem	SF * 6,000,000[1]

Table 12-1. Cardinality of TPC-H Tables

Dbgen code provided by TPC has two options for data generation. In the first option, data is loaded directly to the database through dbgen. The second option is to create raw data in flat files where the “pipe” character separates each field of a record.

In our implementation, we used the second option to generate data. However, in this case we needed to convert the data into “insert” statements compatible with the ORACLE syntax so that we can populate the database through a database client that supports SQL. A separate C++ program is written for this purpose and the ‘insert’ statements are stored in a file for each table. Below is an example of the transformation of a flat file row into an ORACLE insert statement.

```
INSERT INTO supplier VALUES (1,'Supplier#000000001',' N kD4on9OM
Ipw3,gf0JBoQDd7tgrzrddZ',17,'27-918-335-1736',5755.94,'requests haggle care-
fully. accounts sublate finally. carefully ironic pa');
```

Finally, we use ORACLE’s SQLPlus* program to execute the insert statements and populate the database.

WORKLOADS IN PERFORMANCE TEST: As we noted earlier there are 22 defined queries in the TPC-H benchmark. Each of the queries is defined by the four elements:

- a business question
- a functional query definition
- substitution parameters
- a query validation

The business question illustrates the business context in which the query is used. The functional query definition defines the function to be performed by the query using the SQL-92 language. The substitution parameters describe the ways by which the values needed to complete the query syntax may be generated. The query validation describes how to validate the query against the qualification database with SF=1.

Each TPC-H query has one or more substitution parameters. When generating executable query text, a value must be supplied for each substitution parameter. The substitution values should be selected from a uniform distribution over a range or from a list of values specified for each parameter.

TPC provides a query generation utility function “QGEN” to generate executable TPC-H query text. QGEN also generates the randomly substituted variables. In our implementation, we use QGEN to create query text. However we needed to make minor modifications to the query text to conform to the Oracle syntax. We validated the queries against the qualification database of SF=1 by using the default parameter values given in the TPC-H document. The validation queries and their results can be found in Appendix B.

Apart from the 22 queries, The TPC-H benchmark defines two refresh functions as RF1 and RF2. RF1 is called the new sales refresh function. It inserts new rows into the ORDERS and LINEITEM tables in the database following the scaling and data generation methods used to populate the database. RF1 is defined as:

LOOP (SF * 1500) TIMES

    INSERT a new row into the ORDERS table

    LOOP RANDOM(1, 7) TIMES

        INSERT a new row into the LINEITEM table

    END LOOP

END LOOP.

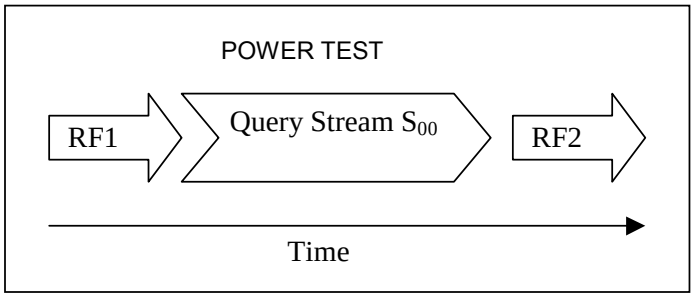


Figure 12-2. Power Test

RF2 is called the old refresh function. It removes old sales information from the database. RF2 is defined as:

LOOP (SF * 1500) TIMES

DELETE FROM ORDERS WHERE O_ORDERKEY = [value]

DELETE FROM LINEITEM WHERE L_ORDERKEY = [value]

END LOOP.

The set of rows to be inserted into the database and the primary key values of the rows to be deleted from the database are created using the *Dbgen* utility. We then convert the data into the appropriate insert and delete statements in the ORACLE syntax.

PERFORMANCE TEST: The performance test consists of two tests:

- power test
- throughput test

The power test must be run followed by a throughput test. We guarantee both sets of test are performed under the same conditions using the same hardware and software configuration and the same data manager and operating system parameters.

Before we delve into the details of the performance test, we give the definitions of the terms that we will use in the subsequent sections of this chapter:

- Q_i represents the i^{th} query of the 22 TPC-H queries.
- A query set is defined as the sequential execution of each and every one of the 22 queries.
- A query stream is defined as the sequential execution of a single query set submitted by a single user.
- *S* (uppercase) represents the number of query streams used during the throughput test.
- *s* (lowercase) represents a given query stream.

A refresh stream is defined as the sequential execution of an integral number of pairs of refresh functions submitted from within a batch program.

A pair of refresh functions is defined as one of each of the two TPC-H refresh functions.

A session is defined as the process context capable of supporting the execution of either a query stream or a refresh stream.

The power test measures the query execution power of the system when connected with a single user. Figure 12-2 depicts the construct of the power test, which consists of the run of a refresh function, *RF*1, followed by a run of a query stream, S_{00}, which is then followed by a run of the delete refresh function, *RF*2. While running the query stream, queries are executed serially.

The throughput test measures the ability of the system to process the most number of queries in the least amount of time. It demonstrates the performance of the selected system against a multi-user workload. The query streams in this test are run in parallel. The number of streams to be run is determined by considering the scale factor (SF) used in a specific system. Table 12-2 shows the relation between the scale

Scale Factor	S
1	2
10	3
100	4
300	6
1,000	7
3,000	8
10,000	9

Table 12-2. Relationship between SF and minimum number of parallel query streams

factor and the minimum number of parallel query streams. In our implementation, the database is populated with a scale factor of 10. We therefore use three different query streams in the throughput test.

The throughput test must also include a refresh stream, which executes pairs of refresh functions serially (i.e., one RF1 followed by one RF2). The number of refresh-function pairs must be equal to the number of query streams used for the throughput test. In scheduling the refresh functions, a given pair must complete before the next pair can be initiated, and within a given pair of refresh functions, RF1 must complete before RF2 can be initiated. Figure 12-3 depicts an implementation of the throughput test.

TPC allows implementations that execute the refresh functions in parallel with the ad-hoc queries as well as systems that segregate query executions from database refreshes. In our implementation, we chose the latter to run the single refresh stream and we executed the stream after the completion of all the query streams.

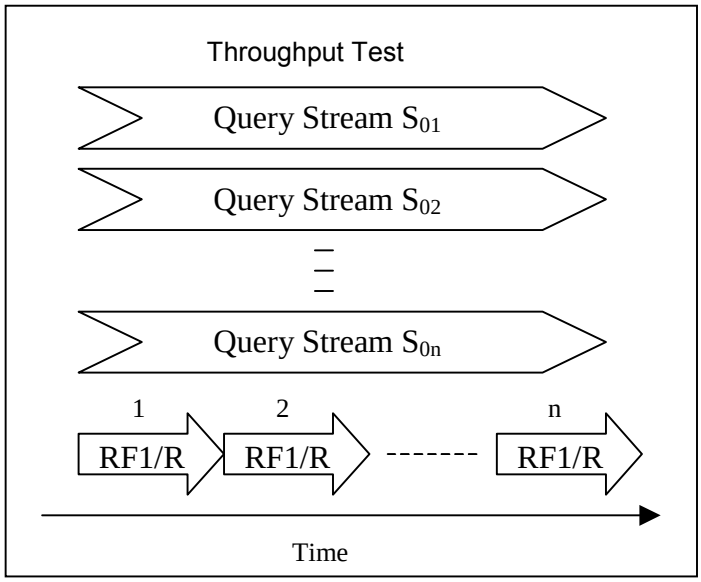


Figure 12-3. Throughput Test

The substitution parameters for each query stream in both the power test and the throughput test must be random. TPC specifies a particular way to seed the random number generator: the time stamp of the database loading time is used as the initial seed. In our implementation, we did not use this seeding method. Instead, we selected a number arbitrarily. Table 12-3 show the seeds used in our implementation of the benchmark. Appendix C shows the substitution parameters used for the streams 00, 01, 02 and 03.

Another issue is the order of the queries within a given stream. Queries should be randomly arranged within a given stream. We use the order sets given in the TPC-H specification document. Table 12-4 presents the query order for each stream.

Seed No	Seeds
Seed0	378994472
Seed1	378994473
Seed2	378994474
Seed3	378994475

Table 12-3. Performance test seeds

Q. No	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
S_{00}	14	2	9	20	6	17	18	8	21	13	3	22	16	4	11	15	1	10	19	5	7	12
S_{01}	21	3	18	5	11	7	6	20	17	12	16	15	13	10	2	8	14	19	9	22	1	4
S_{02}	6	17	14	16	19	10	9	2	15	8	5	22	12	7	13	18	1	4	20	3	11	21
S_{03}	8	5	4	6	17	7	1	18	22	14	9	10	15	11	20	2	21	19	13	16	12	3

Table 12-4. Query Sequences for each stream

12.2 TPC-H Test Results and Analysis

Table 12-5 presents the test results from the TPC-H benchmark experiments. We conducted a total of 63 tests with different storage schemes. The storage schemes we tested include the following:

- *i*-replication (*i* copies of straightforward replication only. When *i*=1, it is similar to the regular NFS case except the file is handled by PASIS FS instead of NFS)
- *i*-replication with crypto (*i* copies of replication, each copy is encrypted)
- *i*-replication with hash (*i* copies of replication. Each copy is accompanied by a hash value for integrity purposes)
- *i*-replication with versioning
- information dispersal
- information dispersal with versioning
- (*n*, *m*) secret sharing with *n* total shares and *m* is the threshold
- RAID 5
- ramp schemes

Tests	**Power Test(second)**	**Throughput Test(second)**	**Query/hour/atsize 1G**
Regular NFS	3548.09	5488.17	35.02
n 1 (Replication)	7350.20	14216.17	15.28
n 2 (Replication)	7854.95	16019.53	13.90
n 3 (Replication)	8579.47	16851.27	13.20
n 4 (Replication)	9125.05	18215.52	12.28
n 5 (Replication)	9497.16	17352.77	12.42
n 6 (Replication)	10046.74	19353.86	11.53
n 7 (Replication)	10346.44	19874.40	11.19
n 8 (Replication)	10907.24	20966.33	10.70
n 1 -c 3 (Replication w. crypto)	9249.03	17703.47	12.23
n 2 -c 3 (Replication w. crypto)	10091.28	18057.62	11.65
n 4 -c 3 (Replication w. crypto)	11659.03	21136.08	10.17
n 8 -c 3 (Replication w. crypto)	14492.26	25142.65	8.62
n 1 -h 2 (Replication w. hash)	8996.95	17406.86	12.55
n 2 -h 2 (Replication w. hash)	9881.43	18446.38	11.73
n 4 -h 2 (Replication w. hash)	11466.20	21243.09	10.25
n 8 -h 2 (Replication w. hash)	14217.50	25271.54	8.68
n 1 -c 3 -q 1 -w 1 (Replication w. vers.)	9320.15	17898.38	12.15
n 2 -c 3 -q 1 -w 2 (Replication w. vers.)	10184.91	18388.23	11.53
n 4 -c 3 -q 1 -w 3 (Replication w. vers.)	13274.83	27092.33	8.34

Tests	Power Test(second)	Throughput Test(second)	Query/hour/atsize 1G
n 4 -c 3 -q 1 -w 4 (Replication w. vers.)	11481.69	21408.95	10.11
n 8 -c 3 -q 1 -w 5 (Replication w. vers.)	19549.86	36074.21	5.98
n 8 -c 3 -q 1 -w 6 (Replication w. vers.)	17981.07	32699.06	6.63
n 1 -m 1 (systematic IDA)	7713.89	14983.45	14.58
n 2 -m 1 (systematic IDA)	8567.61	16744.01	13.21
n 2 -m 2 (systematic IDA)	7358.71	13877.85	15.63
n 4 -m 1 (systematic IDA)	9493.05	18420.50	12.03
n 4 -m 2 (systematic IDA)	8165.43	15460.44	14.12
n 4 -m 4 (systematic IDA)	7294.62	12976.15	16.36
n 8 -m 1 (systematic IDA)	11946.24	23053.48	9.78
n 8 -m 2 (systematic IDA)	9102.75	17094.93	12.73
n 8 -m 4 (systematic IDA)	8373.67	14144.53	14.86
n 8 -m 8 (systematic IDA)	7060.95	12603.95	17.16
n 8 -m 4 -q 1 -w 5 (IDA w. versioning)	8980.10	15331.69	13.65
n 8 -m 4 -q 1 -w 6 (IDA w. versioning)	8927.86	15539.01	13.56
n 8 -m 4 -q 1 -w 7 (IDA w. versioning)	8987.77	15469.89	13.60
n 8 -m 4 -q 1 -w 8 (IDA w. versioning)	8965.64	15582.38	13.57
n 8 -m 5 -q 1 -w 5 (IDA w. versioning)	8729.70	14954.86	14.12
n 2 (RAID5)	7931.57	15348.59	14.28
n 4 (RAID5)	7409.78	12928.37	16.38
n 8 (RAID5)	7369.04	12626.09	16.91
n 1 -m 1 (Secret Sharing)	8432.89	16367.01	13.34
n 2 -m 1 (Secret Sharing)	9090.68	16582.42	12.79
n 2 -m 2 (Secret Sharing)	12980.68	24756.13	8.77
n 4 -m 1 (Secret Sharing)	9953.30	19111.04	11.44
n 4 -m 2 (Secret Sharing)	14474.94	25575.73	8.30
n 4 -m 4 (Secret Sharing)	22801.56	40824.35	5.30
n 8 -m 1 (Secret Sharing)	12572.93	23826.81	9.38
n 8 -m 2 (Secret Sharing)	17173.37	31875.60	6.94
n 8 -m 4 (Secret Sharing)	26404.37	48429.75	4.58
n 8 -m 8 (Secret Sharing)	43761.12	81086.84	2.78
n 8 -m 4 -p 1 (Ramp)	9194.85	16438.60	12.88
n 8 -m 4 -p 2 (Ramp)	11618.79	20342.24	10.42
n 8 -m 4 -p 3 (Ramp)	14822.44	25824.56	8.19
n 8 -m 6 -p 1 (Ramp)	9364.03	16885.11	12.73
n 8 -m 6 -p 2 (Ramp)	10468.11	18262.01	11.44
n 8 -m 6 -p 4 (Ramp)	16042.76	27720.02	7.61
n 8 -m 6 -p 3 (Ramp)	12212.97	21410.40	9.80

Table12-5. TPC-H benchmark results

Power test result analysis: Figure 12-4 plots the power test results of the various replication schemes. Note that we did not include the regular NFS performance in the comparison. The reason for this is that

the decrease in performance, from 3548ms of NFS to 7350ms of 1-replication, is due to the particular implementation of our prototype system. For instance, our prototype employs an NFS looping back mechanism to invoke PASIS FS, and in the end again emulates an NFS client to invoke the backend storage. A different and more efficient implementation, which will not involve emulating NFS multiple times, could eradicate this performance penalty. The more interesting comparison is the different storage schemes against the 1-replication case, which represents a conventional storage service with no survivability mechanisms. Here forth we will use the 1-replication case as our base case for analysis purposes.

As shown in Figure 12-4, the performance penalty associated with replication is largely linear with respect to the number of replications. This is expected as the power test involves two refresh streams consisting of write operations. When *n* increases, the cost associated with writes also increases. The increase in execution time is mostly due to the write cost (read remains the same for straight replication). Performance degrades linearly with increased degree of replication.

Figure 12-4 also shows that the slope of increase is a gentle one: the increase in the execution time is not drastic. From 1-replication to 8-replication, the increase is 47%.

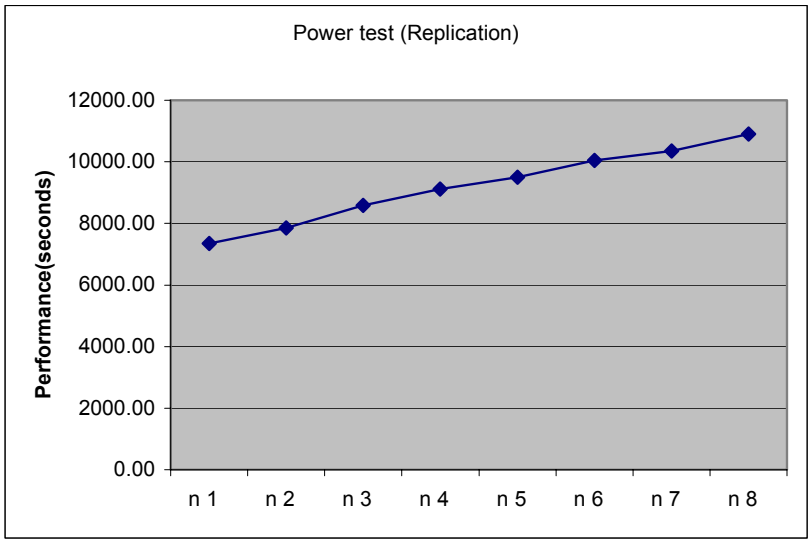


Figure 12-4. Power test (Replication only)

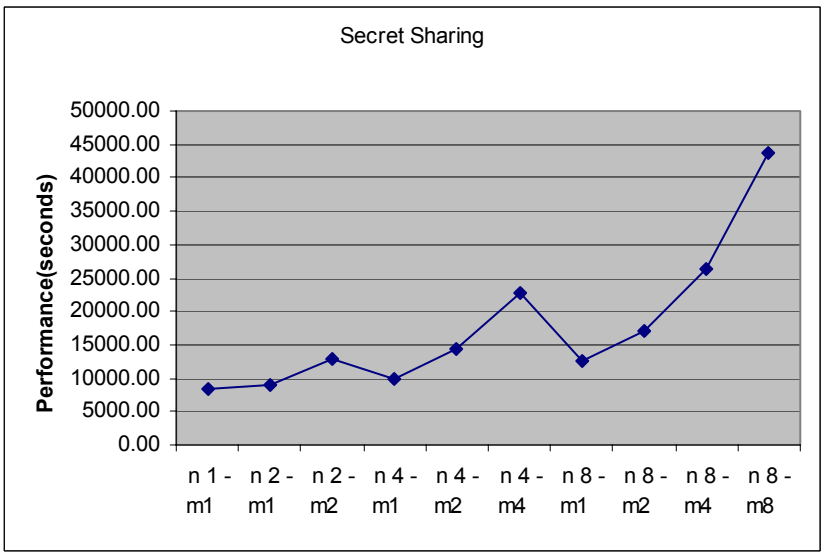


Figure 12-5. Power test (secret sharing)

Figure 12-5 shows the power test results of secret sharing schemes with an increasing *n* (i.e., the number of shares written out.) Note that secret sharing with *n*=1 and *m*=1 is similar to the case of 1-replication. The difference between the two lies in when the Encode/Decode module is invoked; 1-replication executes the replication function while the secret sharing scheme executes the secret sharing function. As shown, the secret sharing encode and decode functions are more expensive than those of replication.

Figure 12-5 also shows that when *m* increases while *n* stays constant, the cost associated with secret sharing also increases linearly. Note when *m* increases, we observe a much sharper increase slope. This is because the power test is dominated by reads. Therefore increasing the *m* value has more significant impact on the performance than increasing *n*.

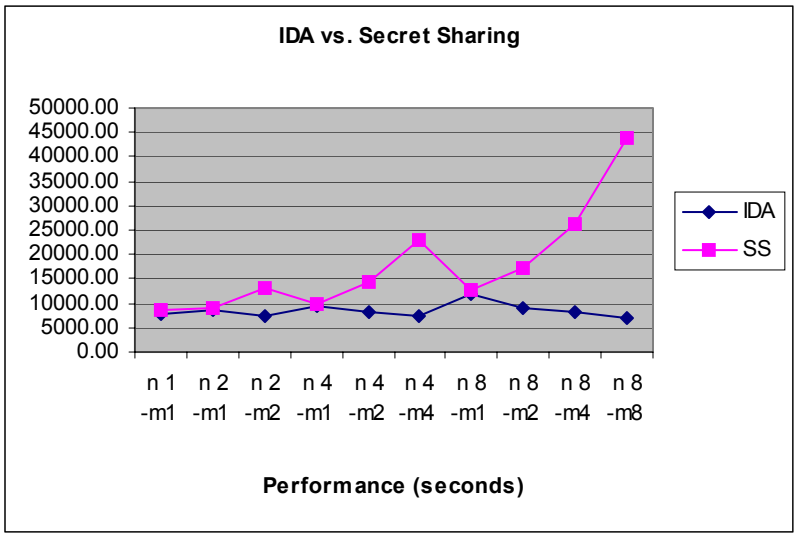


Figure 12-6. Power test (IDA vs. SS)

Next we compare the performance of IDA with secret sharing. The comparisons are shown in Figure 12-6. It is clear that the cost for IDA is mainly associated with the value of *n.* An increasing *m* has the opposite effect on performance—the larger the *m* is (assuming *n* stays constant), the more efficient the system is. The comparisons in Figure 12-6 show that for schemes with large *n* and *m* values, IDA is clearly more efficient than secret sharing.

Throughput result analysis: We use the results of the power test and the throughput test to derive the true throughput, represented as the number of 1GB queries/hour. This data is shown in the query/hour column in Table 12-5.

Figure 12-7 shows the throughput numbers with replication. We can see that the replication throughput declines only slightly as the number of copies, *n,* increases. These results show that replication, in itself, does not have any significant adverse effect on the system performance. It should be noted that while in theory, more replicated copies can lead to better performance, our prototype implementation includes no optimization technique, performance tuning, or load balancing. The performance results we observed with the prototype implementation do not represent the full potential of the data distribution schemes.

Figure 12-8 shows the throughput results of IDA and secret sharing. When *m*=1, the behavior of IDA is very similar to SS. But when the value of *m* increases, IDA and Secret sharing mechanisms become more disparate from each other—the throughput of secret sharing decreases while the throughput of IDA increases. This is partially because in secret sharing, the cost of both reads and writes (encode and decode cost) increase dramatically with *m*, while for IDA, the cost of reads and writes stays very much the same.

12.3 Summary

TPC-H is a typical workload with very few writes and primarily non-I/O operations. Such a workload is representative of the database workload that is likely to arise in a PASIS system. Comparing with the more I/O intensive workload in the trade-off analysis section (Chapter 7), we observed a significant reduction in penalty for the TPC-H

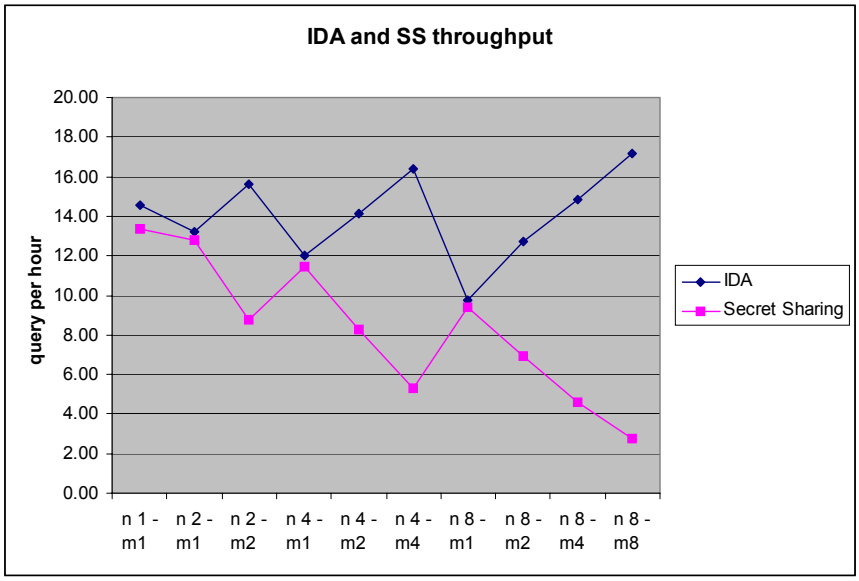


Figure 12-7. Throughput for IDA and SS

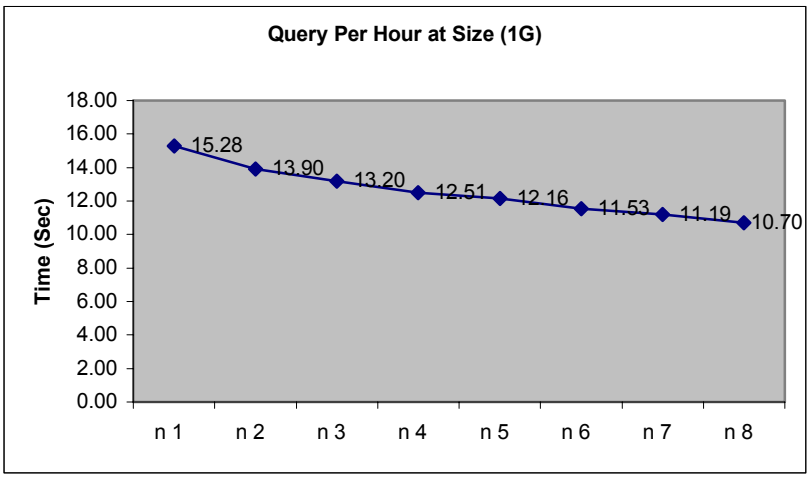


Figure 12-8. Throughput for Replication

workload. For instance, the exponential drop in performance experienced by the various storage schemes (against 1-replication) in the default configuration is reduced to a near linear drop in performance with the PASIS-like workload. Though in our experiments, we have not separated the I/O-related performance penalty with the non-I/O penalty, we are reasonably confident that the impact of reducing I/O performance is substantially limited in a PASIS environment. PASIS-enhanced storage can make significant contribution to the survivability of PASIS without considerably damaging the overall performance.

13 REFERENCES

[Adya95] Atul Adya, Robert Gruber, Barbara Liskov, and Umesh Maheshwari. Efficient optimistic concurrency control using loosely synchronized clocks. ACM SIGMOD International Conference on Management of Data, pages 23–34, 1995.

[Adya97] Atul Adya and Barbara Liskov. Lazy consistency using loosely synchronized clocks. ACM Symposium on Principles of Distributed Computing (Santa Barbara, CA, August 1997), pages 73–82. ACM, 1997.

[Alexandrov97] A. Alexandrov, M. Ibel, et al. Extending the Operating System at the User Level: the UFO Global File System. Proceedings of the 1997 USENIX Annual Technical Conference, Anaheim, CA, January 1997.

[Amir96] Y. Amir, A. Wool "Evaluating quorum systems over the Internet", Proceedings of the Twenty-Sixth Annual International Symposium on Fault-Tolerant Computing, 1996

[Amir98] Y. Amir, A. Wool "Optimal Availability Quorum Systems: Theory and Practice", Information Processing Letters, 1998

[Amiri 99] Khalil Amiri, Garth A. Gibson, and Richard Golding. Scalable concurrency control and recovery for shared storage arrays. CMU–CS–99–111. Computer Science Department, Carnegie-Mellon University, Pittsburgh, PA, February 1999.

[Amiri00] Khalil Amiri, Garth A. Gibson, and Richard Golding. Highly concurrent shared storage. International Conference on Distributed Computing Systems (Taipei, Taiwan, 10–13 April 2000), pages 298–307. IEEE Computer Society, 2000.

[AndersonD00] Darrell C. Anderson, Jemrey S. Chase, and Amin M. Vahdat. Interposed request routing for scalable network storage. Symposium on Operating Systems Design and Implementation (San Diego, CA, 22-25 October 2000), 2000.

[AndersonR96] Ross J. Anderson. The Eternity Service. PRAGOCRYPT, pages 242-253. CTU Publishing, 1996.

[AndersonT96] Thomas E. Anderson, Michael D. Dahlin, Jeanna M. Neefe, David A. Patterson, Drew S. Roselli, and Randolph Y.Wang. Serverless network file systems. ACM Transactions on Computer Systems, 14 (1):41-79, February 1996.

[Baker91] Mary G. Baker, John H. Hartman, Michael D. Kupfer, Ken W. Shirriff, and John K. Ousterhout. Measurements of a distributed file system. ACM Symposium on Operating System Principles (Asilomar, Pacific Grove, CA). Published as Operating Systems Review, 25(5):198–212, 13–16 October 1991.

[Bakkaloglu02] Mehmet Bakkaloglu, Jay J. Wylie, Chenxi Wang, Gregory R. Ganger. On Correlated Failures in Survivable Storage Systems. CMU SCS Technical Report CMU-CS-02-129. May 2002.

[Bakkaloglu02a] M. Bakkaloglu, J. J. Wylie, C. Wang, and G. R. Ganger. Correlated failures in survivable storage systems. In the fast abstract proceedings of the 2002 IEEE/IFIP Dependable Systems and Networks (DSN'02). Bethesda, MD. June 2002.

[Becker96] B. Becker, S. Gschwind, T. Ohler, P. Widmayer, and B. Seeger. An asymptotically optimal multiversion b-tree. Very large data bases journal, 5(4):264–275, 1996.

[Berlekamp68] Elwyn Berlekamp. Algebraic coding theory. McGraw-Hill, New York, 1968.

[Berlekamp84] E.R. Berlekamp, Algebraic Coding Theory, Second Edition, Aegean Park Press, Laguna Hills, California, 1984.

[Bigrigg01] Michael W. Bigrigg, and Joseph G. Slember. Testing the portability of desktop applications to a networked embedded system. Workshop on Reliability in Embedded Systems at the 20th IEEE Symposium on Reliable Distributed Systems, New Orleans, October 2001.

[Bigrigg02a] M. Bigrigg and J. Vos. The Set-Check-Use Methodology for Detecting Error Propagation Failures in I/O Routines. Workshop on Dependability Benchmarking in conjunction with The International Conference on Dependable Systems and Networks, Washington, DC, June 2002.

[Bigrigg02b] Michael W. Bigrigg. Robustness Hinting for Improving End-to-End Dependability Second Workshop on Evaluating and Architecting System Dependability (EASY), in conjunction with ASPLOS-X, October 2002, San Jose, California, U.S.A.

[Blackwell95] Trevor Blackwell, Jeffrey Harris, and Margo Seltzer. Heuristic cleaning algorithms in log-structured file systems. Annual USENIX Technical Conference (New Orleans), pages 277-288. Usenix Association, 16-20 January 1995.

[Blahut83] R.E. Blahut, Theory and Practice of Error-Control Codes, Addison-Wesley, Reading, Massachusetts, 1983.

[Blakley79] G. R. Blakley. Safeguarding cryptographic keys. AFIPS National Computer Conference (New York, NY, 4-7 June 1979), pages 313-317. AFIPS, 1979.

[Blakley85] G. R. Blakley, C. Meadows "Security of Ramp Schemes", Advances in Cryptology – CRYPTO, pages 242-268, Springer-Verlag 1985

[Bolosky00a] William J. Bolosky, Scott Corbin, David Goebel, and John R. Douceur. Single Instance Storage in Windows 2000. USENIX Windows Systems Symposium, Seattle, WA, 3-4 August 2000, pages 13-24. USENIX Association, 2000.

[Bolosky00b] William J. Bolosky, John R. Douceur, David Ely, and Marvin Theimer. Feasibility of a serverless distributed file system deployed on an existing set of desktop PCs. ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, Santa Clara, CA, 17-21 June 2000. Published as Performance Evaluation Review, 28 (1):34-43. ACM, 2000.

[Broadwell02] P. Broadwell, N. Sastry, and J. Traupman. FIG: A Prototype Tool for Online Verification of Recovery Mechanisms. Workshop on Self-Healing, Adaptive, and self-MANaged Systems (SHAMAN), New York, NY, June 2002.

[Brown00] A. Brown, and D. Patterson. Towards Availability Benchmarks: A Case Study of Software RAID Systems. Proceedings of the 2000 USENIX Annual Technical Conference, San Diego, CA, June 2000.

[Burns96] Randal C. Burns. Differential compression: a generalized solution for binary files. Masters thesis. University of California at Santa Cruz, December 1996.

[Burrows92] Michael Burrows, Charles Jerian, Butler Lampson, and Timothy Mann. On-line data compression in a log-structured file system. Architectural Support for Programming Languages and Operating Systems (Boston, MA, 12-15 October 1992). Published as Computer Architecture News, 20(special issue):2-9, October 1992.

[Burrows94] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. 124. Digital Equipment Corporation Systems Research Center, Palo Alto, CA, 10 May 1994.

[Castro98] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. Symposium on Operating Systems Design and Implementation (New Orleans, LA, 22–25 February 1999), pages 173–186. ACM, 1998.

[Chakrabarti02] A. Chakrabarti, L. de Alfaro, et. al. Interface Compatibility Checking for Software Modules. Proceedings of the 14th International Conference on Computer-Aided Verification (CAV), Lecture Notes in Computer Science 2404, 2002.

[Cristian95] F. Cristian. Exception Handling and Tolerance of Software Faults. Software Fault Tolerance, Michael R. Luy (Ed.). Chichester: Wiley, 1995.

[Dai] Wei Dai. Crypto++ reference manual. http://cryptopp.sourceforge.net/docs/ref/.

[Dasgupta99] P. Dasgupta, V. Karamcheti, and Z. Kedem, Transparent Distribution Middleware for General Purpose Computations. Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'99), June 1999.

[Deepinder94] Deepinder S. Gill, Songian Zhou, and Harjinder S. Sandhu. A case study of file system workload in a large–scale distributed environment. Technical report CSRI–296. University of Toronto, Ontario, Canada, March 1994.

[Denning87] Dorothy Denning. An intrusion-detection model. IEEE Transactions on Software Engineering, SE-13(2):222-232, February 1987.

[Deswarte91] Yves Deswarte, L. Blain, and Jean-Charles Fabre. Intrusion tolerance in distributed computing systems. IEEE Symposium on Security and Privacy, Oakland, CA, 20-22 May 1991, pages 110-121, 1991.

[DeVale01] J. DeVale and P. Koopman. Performance Evaluation of Exception Handling in I/O Libraries. The Proceedings of the International Conference on Dependable Systems and Networks, Goteborg, Sweden, June 2001.

[DeWitt90] S. Ghandeharizadeh, D. Schneider, H. Hsiao, A. Bricker, R. Rasmussen "The GAMMA Database Machine Project", IEEE Transactions on Knowledge and Data Engineering, Vol. 2, No. 1, March 1990.

[Douceur01a] J. R. Douceur, R. P. Wattenhofer "Modeling Replica Placement in a Distributed File System: Narrowing the Gap between Analysis and Simulation", Proceedings of 9th ESA,, pp. 356-367, 2001.

[Douceur01b] J. R. Douceur, R. P. Wattenhofer "Competitive Hill-Climbing Strategies for Replica Placement in a Distributed File System", Proceedings of 15th DISC, pp. 48-62, 2001.

[Douceur01c] J. R. Douceur, R. P. Wattenhofer. Optimizing File Availability in a Secure Serverless Distributed File System. Proceedings of 20th IEEE SRDS, pp. 4-13, 2001.

[Ellard03] D. Ellard, J. Ledlie, P. Malkani, and M. Seltzer. Passive NFS Tracing of an Email and Research Workload. Conference on File and Storage Technologies. USENIX Association, 2003.

[Engler00] D. Engler, B. Chelf, et al. Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions. The 4th Symposium on Operating Systems Design and Implementation, San Diego, CA, October 2000.

[Engler01] D. Engler, D. Chen, et al. Bugs as Deviant Behavior: A General Approach to Inferring Errors in System Code. Symposium on Operating System Principles (SOSP'01), Banff, Canada, October 2001.

[Evans94] D. Evans, J. Guttag, et al. LCLint: A Tool for Using Specifications to Check Code. SIGSOFT Symposium on the Foundations of Software Engineering, December 1994.

[eVault] eVault. http://www.bell-labs.com/user/garay/#projects.

[Farsite] Farsite. http://www.research.microsoft.com/sn/Farsite/.

[Fault81] The Fault tree handbook. U.S. Nuclear Regulatory Commission Report NUREG-0492.

[Feldman87] P. Feldman, A Practical Scheme for Non-interactive Verifiable Secret Sharing. In Proc. 28th IEEE Symposium of the Foundation of Computer Science, pages 427-437, 1987.

[FreeHaven] FreeHaven. http://www.freehaven.net/.

[Ganger01] Survivable Storage Systems. Gregory R. Ganger, Pradeep K. Khosla, Mehmet Bakkaloglu, Michael W. Bigrigg, Garth R. Goodson, Semih Oguz, Vijay Pandurangan, Craig A. N. Soules, John D. Strunk, Jay J. Wylie. DARPA Information Survivability Conference and Exposition (Anaheim, CA, 12-14 June 2001), pages 184-195 vol 2. IEEE, 2001

[Garg99] S. Garg,, Huang, Y., Kintala, C. M. R. Kintala, K. S. Trivedi, S. Yajnik. "Performance and Reliability Evaluation of Passive Replication Schemes in Application Level Fault Tolerance", Proc. Twenty-Ninth International Symposium on Fault-Tolerant Computing (FTCS-29), June 1999

[Gibson 98] Garth A. Gibson, David F. Nagle, Khalil Amiri, Jeff Butler, Fay W. Chang, Howard Gobioff, Charles Hardin, Erik Riedel, David Rochberg, and Jim Zelenka. A cost-effective, high-bandwidth storage architecture. Architectural Support for Programming Languages and Operating Systems, San Jose, CA, 3-7 October 1998. Published as SIGPLAN Notices, 33 (11):92-103, November 1998.

[Gibson99] Garth A. Gibson, David F. Nagle, William Courtright II, Nat Lanza, Paul Mazaitis, Marc Unangst, and Jim Zelenka. NASD scalable storage systems. USENIX.99 (Monterey, CA., June 1999), 1999.

[Gladman] Brian Gladman. AES implementation. http://fp.gladman.plus.com/cryptography_technology/rijndael/index.htm.

[Goldberg98] [Goldberg98] [4]Andrew V. Goldberg and Peter N. Yianilos. Towards an archival intermemory. IEEE International Forum on Research and Technology Advances in Digital Libraries, Santa Barbara, CA, 22-24 April 1998, pages 147-156. IEEE, 1998.

[Golding99] Richard Golding and Elizabeth Borowsky. Fault-tolerant replication management in large-scale distributed storage systems. Symposium on Reliable Distributed Systems (Lausanne, Switzerland, 19–22 October 1999), pages 144–155. IEEE Computer Society, 1999.

[Gong89] Li Gong. Securely replicating authentication services. International Conference on Distributed Computing Systems (Newport Beach, CA), pages 85–91. IEEE Computer Society Press, 1989.

[Goodenough75] J. Goodenough. Exception Handling: Issues and a Proposed Notation. Communications of the Association of Computing Machinery. December 1975.

[Goodson02] Garth R. Goodson, Jay J. Wylie, Gregory R. Ganger, and Michael K. Reiter. Decentralized storage consistency via versioning servers. Technical Report CMU–CS–02–180. September 2002.

[Goodson03a] Byzantine-tolerant Erasure-coded Storage. Garth R. Goodson, Jay J. Wylie, Gregory R. Ganger, Michael K. Reiter. Carnegie Mellon University Technical Report CMU-CS-03-187, September 2003.

[Goodson03b] Efficient Consistency for Erasure-coded Data via Versioning Servers. Garth R. Goodson, Jay J. Wylie, Gregory R. Ganger, Michael K. Reiter. Carnegie Mellon University Technical Report CMU-CS-03-127, April 2003.

[Goodson03c] A protocol family for versatile survivable storage infrastructures Garth R. Goodson, Jay J. Wylie, Gregory R. Ganger, Michael K. Reiter CMU-PDL-03-103, Dec. 03

[Gray78] J. N. Gray. Notes on data base operating systems. In , volume 60, pages 393–481. Springer-Verlag, Berlin, 1978.

[Gray91] Jim Gray and Daniel P. Siewiorek. High-availability computer systems. IEEE Computer, 24 (9):39-48, September 1991.

[Griffiths73] D. A. Griffiths. Maximum Likelihood Estimator for the Beta-Binomial Distribution and an Application to the Household Distribution of the Total Number of Cases of a Disease. Biometrics vol. 29, pp. 637-648, December 1973.

[Grune] D. Grune, B. Berliner, and J. Polk. Concurrent Versioning System, http://www.cvshome.org/.

[Hagmann87] Robert Hagmann. Reimplementing the Cedar file system using logging and group commit. ACM Symposium on Operating System Principles (Austin, Texas, 8-11 November 1987). Published as Operating Systems Review, 21(5):155-162, November 1987.

[Hansen] Eric Hansen. Lecture 19 notes: Goodness of Fit Tests. Dartmouth University Engs27 class notes.

[Hardin68] Garrett Hardin. "The Tragedy of the Commons." Science, 162 (1968) pages 1243-1248.

[Herlihy90] Maurice P. Herlihy and Jeanette M. Wing. Linearizability: a correctness condition for concurrent objects. ACM Transactions on Programming Languages and Systems, 12(3):463–492. ACM, July 1990.

[Herzberg95] Amir Herzberg, Stanislaw Jarecki, Hugo Krawczyk, Moti Yung, Proactive Secret Sharing Or: How to Cope With Perpetual Leakage. Lecture notes in Computer Science, 1995.

[Hitz94] David Hitz, James Lau, and Michael Malcolm. File system design for an NFS file server appliance. Winter USENIX Technical Conference (San Francisco, CA). Published as Proceedings of USENIX, pages 235-246. USENIX Association, 19 January 1994.

[Howard88] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and performance in a distributed file system. ACM Transactions on Computer Systems, 6(1):51-81, February 1988.

[Hunt99] G. Hunt and D. Brubacher. Detours: Binary Interception of Win32 Functions. Proceedings of the 3rd USENIX Windows NT Symposium. Seattle, WA, July 1999.

[Intermemory] Intermemory. http://www.intermemory.org/.

[Iyengar98] A. Iyengar, R. Cahn, C. Jutla, and J. A. Garay. Design and implementation of a secure distributed data repository. IFIP International Information Security Conference, Vienna, Austria, and Budapest, Hungary, 31 August-4 September 1998, pages 123-135. ACM, 1998.

[Jacobson88] V. Jacobson. “Congestion avoidance and control.” Proceedings of ACM SIGCOMM 1988. pp 314-329. August 1988

[Jain91] Raj Jain. The art of computer systems performance analysis. John Wiley & Sons, 1991.

[Johnson96] James E. Johnson and William A. Laing. Overview of the Spiralog file system. Digital Technical Journal, 8(2):5-14, 1996.

[Kalyanakrishnam99] M. Kalyanakrishnam, Z. Kalbarczyk, R. Iyer "Failure Data Analysis of LAN of Windows NT Based Computers", Proc. of 18th Symposium on Reliable and Distributed Systems, SRDS '99, Lausanne, Switzerland, pp.178-187, 1999

[Karn87] Phil Karn and Craig Pardridge. Improving Round-Trip Time Estimates in Reliable Transport Protocols. ACM SIGCOMM '87 (Aug. 1987).

[Katcher97] Jeffrey Katcher. PostMark: a new file system benchmark. TR3022. Network Appliance, October 1997.

[Kim01] Minkyong Kim and Brian Noble. Mobile Network Estimation. Proceedings of the ACM Conference on Mobile Computing and Networking, Rome, Italy. June 2001.

[Kim94] Gene H. Kim and Eugene H. Spafford. The design and implementation of Tripwire: a file system integrity checker. Conference on Computer and Communications Security (Fairfax, Virginia), pages 18-29, 2-4 November 1994.

[Kistler92] James J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. ACM Transactions on Computer Systems, 10(1):3–25. ACM Press, February 1992.

[Kleinrock73] Leonard Kleinrock. Queuing systems. John Wiley and Sons, 1973.

[Knuth81] Donald Ervin Knuth. Seminumerical algorithms, volume 2. Addison-Wesley, 1981.

[Krawczyk93] Hugo Krawczyk. Distributed fingerprints and secure information dispersal. ACM Symposium on Principles of Distributed Computing (Ithaca, NY, 15–18 August 1993), pages 207–218, 1993.

[Krawczyk94] Hugo Krawczyk. Secret sharing made short. Advances in Cryptology -CRYPTO (Santa Barbara, CA, 22–26 August 1993), pages 136–146. Springer-Verlag, 1994.

[Kubiatowicz00] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaten, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells, and Ben Zhao. OceanStore: an architecture for global-scale persistent storage. Architectural Support for Programming Languages and Operating Systems (Cambridge, MA, 12–15 November 2000). Published as Operating Systems Review, 34(5):190–201, 2000.

[Lang98] J. Lang and D. Stewart. A Study of the Applicability of Existing Exception-Handling Techniques to Component-Based Real-Time Software Technology. ACM Transactions on Programming Languages and Systems, Volume 20, March 1998.

[Lee83] P. Lee. Exception Handling in C Programs. Software-Practice and Experience, Volume 13, 1983.

[Lee96] Edward K. Lee and Chandramohan A. Thekkath. Petal: distributed virtual disks. Architectural Support for Programming Languages and Operating Systems, Cambridge, MA, 1-5 October 1996. Published as SIGPLAN Notices, 31 (9):84-92, 1996.

[Liskov91] Barbara Liskov, Sanjay Ghemawat, Robert Gruber, Paul Johnson, Liuba Shrira, and Michael Williams. Replication in the Harp file system. ACM Symposium on Operating System Principles (Pacific Grove, CA, 13–16 October 1991). Published as Operating Systems Review, 25(5):226–238, 1991.

[Littlewood89] B. Littlewood, D.R. Miller, “Conceptual modeling of coincident failures in multiversion software”, IEEE Transactions on Software Engineering, Volume: 15 Issue: 12, Pages: 1596-1614, Dec. 1989

[Luby01] Michael G. Luby, Michael Mitzenmacher, M. Amin Shokrollahi, and Daniel A. Spielman. Efficient Erasure Correcting Codes. IEEE Transactions on Information Theory, 47(2):569–584. IEEE, February 2001.

[Luby98] M. Luby, M. Mitzenmacher, M. Shokrollahi, D. Spielman, V. Stemann “Analysis of low density codes and improved designs using irregular graphs”, In Proc. Of ACM STOC, May 1998

[Lumb00] Christopher Lumb, Jiri Schindler, Gregory R. Ganger, David F. Nagle, and Erik Riedel. Towards higher disk head utilization: Extracting \free" bandwidth from busy disk drives. Symposium on Operating Systems Design and Implementation (San Diego, CA, 23-25 October 2000). ACM, October 2000.

[MacDonald00] Josh MacDonald. File system support for delta compression. Masters thesis. Department of Electrical Engineering and Computer Science, University of California at Berkeley, 2000.

[MacDonald98] Josh MacDonald, Paul N. Hilffinger, and Luigi Semenzato. PRCS: The project revision control system. European Conference on Object-Oriented Programming (Brussels, Belgium, July, 20-21). Published as Proceedings of ECOOP, pages 33-45. Springer-Verlag, 1998.

[Malkhi00] D. Malkhi, M. Reiter, A. Wool “The load and availability of Byzantine quorum systems”, SIAM J. Computing, 29(6):1889-1906, 2000

[Malkhi00a] D. Malkhi, and M. Reiter. An Architecture for Survivable Coordination in Large Distributed Systems. IEEE Transactions on Knowledge and Data Engineering, April 2000, 12(2):187—202.

[Malkhi98] Dahlia Malkhi and Michael Reiter. Byzantine Quorum Systems. Distributed Computing, 11(4):203–213. Springer–Verlag, 1998.

[Martin02] Jean-Philippe Martin, Lorenzo Alvisi, and Michael Dahlin. Minimal byzantine storage. International Symposium on Distributed Computing (Toulouse, France, 28–30 October 2002), 2002.

[Matthews97] Jeanna Neefe Matthews, Drew Roselli, Adam M. Costello, Randolph Y. Wang, and Thomas E. Anderson. Improving the performance of log-structured file systems with adaptive methods. ACM Symposium on Operating System Principles (Saint-Malo, France, 5-8 October 1997). Published as Operating Systems Review, 31(5):238-252. ACM, 1997.

[McCoy90] K. McCoy. VMS file system internals. Digital Press, 1990.

[McKusick84] Marshall K. McKusick, William N. Joy, Samuel J. Lefer, and Robert S. Fabry. A fast file system for UNIX. ACM Transactions on Computer Systems, 2(3):181-197, August 1984.

[Melliar-Smith77] P. Melliar-Smith and B. Randell, Software Reliability: The Role of Programmed Exception Handling. Proceedings of the Conference on Language Design for Reliable Software, SIGPLAN Notice, Volume 12(3), March 1977.

[Menezes97] A. Menezes, P. van Oorschot and S. Vanstone. Handbook of Applied Cryptography. CRC Press LLC, USA, 1997.

[Miller90] B. Miller, L. Fredriksen, B. So. An Empirical Study of the Reliability of UNIX Utilities. Communications of the Association for Computing Machinery. December 1990.

[Mills92] David L. Mills. Network time protocol (version 3), RFC–1305. IETF, March 1992.

[Mills95] David L. Mills. Improved algorithms for synchronizing computer network clocks. IEEE-ACM Transactions on Networking, 3(3), June 1995.

[Muthitacharoen01] A. Muthitacharoen, B. Chen, and D. Mazieres. A low-bandwidth network file system. ACM Symposium on Operating System Principles. Published as Operating System Review, 35(5):174–187. ACM, 2001.

[Nicola90] V. F. Nicola, A. Goyal. Modeling of Correlated Failures and Community Error Recovery in Multiversion Software. IEEE Transaction on Software Engineering, Vol. 16 No.3, March 1990.

[Noble94] Brian D. Noble and M. Satyanarayanan. An empirical study of a highly available file system. ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (Nashville, TN, 16–20 May 1994). Published as Performance Evaluation Review, 22(1):138–149. ACM, 1994.

[OBSD99] Object based storage devices: a command set proposal. Technical report. October 1999. http://www.T10.org/.

[Ortalo99] Rodolphe Ortalo, Yves Deswarte, and Mohamed Kaaniche. Experimenting with quantitative evaluation tools for monitoring operational security. IEEE Transactions on Software Engineering, 25 (5):633{650. IEEE, September 1999.

[PASIS] PASIS, http://www.ices.cmu.edu/pasis.

[Pandurangan02] Vijay Pandurangan. On Modeling and Improving the Performance of Threshold Scheme-Based Distributed Storage Systems. M.Sc. Thesis, Carnegie Mellon University, Pittsburgh, PA, USA. July 19, 2002.

[PattersonD88] D. Patterson, G. Gibson, and R. Katz, "A case for redundant arrays of inexpensive disks (RAID)," Proceedings of International Conference on Management of Data, June 1988, pages 109-116.

[Pennington02] Adam G. Pennington, John D. Strunk, John Linwood Griffin, Craig A. N. Soules, Garth R. Goodson, and Gregory R. Ganger. Storage-based intrusion detection: Watching storage activity for suspicious behavior. Technical report CMU–CS–02–179. Carnegie Mellon University, October 2002.

[Plank97] J. Plank "A tutorial on Reed-Solomon coding for fault-tolerance in raid-like systems", Software Practice and Experience, 27(9):995-1012, September 1997

[Plauger92] P. Plauger. The Standard C Library. Prentice Hall, 1992.

[Poess00] M. Poess, C. Floyd, New TPC Benchmarks for Decision Support and Web Commerce, SIGMOD Record, Vol. 29, No.4, December 2000.

[Powell00] A.L.Powell, J.C.French, "Growth and Availability of the NCSTRL Digital Library", 1-581 13-231, ACM 2000.

[Quinlan02] S. Quinlan and S. Dorward. Venti: a new approach to archival storage. Conference on File and Storage Technologies, pages 89–101. USENIX Association, 2002.

[Rabin89] M. O. Rabin. Efficient Dispersal of Information for Security, Load Balancing and Fault Tolerance. Journal of the ACM, 36(2):335-348 ACM April 1989.

[Rosenblum91] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. ACM Symposium on Operating System Principles. Published as Operating Systems Review, 25(5):1–15, 1991.

[Rosenblum92] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. ACM Transactions on Computer Systems, 10(1):26-52, February 1992.

[Santry99] Douglas S. Santry, Michael J. Feeley, Norman C. Hutchinson, Ross W. Carton, Jacob Ofir, and Alistair C. Veitch. Deciding when to forget in the Elephant file system. ACM Symposium on Operating System Principles (Kiawah Island Resort, South Carolina). Published as Operating Systems Review, 33(5):110-123. ACM, 12-15 December 1999.

[Schneider90] [53] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. ACM Computing Surveys, 22(4):299–319, December 1990.

[Schneier00] Bruce Schneier. Secrets and Lies: Digital Security in a Networked World. John Wiley, 2000.

[Schneier96] B. Schneier. Applied Cryptography 2nd Ed. John Wiley & Sons, USA, 1996.

[Seltzer00] Margo I. Seltzer, Gregory R. Ganger, M. Kirk McKusick, Keith A. Smith, Craig A. N. Soules, and Christopher A. Stein. Journaling versus Soft Updates: Asynchronous Meta-data Protection in File Systems. USENIX Annual Technical Conference (San Diego, CA), 18-23 June 2000.

[Seltzer95] Margo Seltzer, Keith A. Smith, Hari Balakrishnan, Jacqueline Chang, Sara McMains, and Venkata Padmanabhan. File system logging versus clustering: a performance comparison. Annual USENIX Technical Conference (New Orleans), pages 249-264. Usenix Association, 16-20 January 1995.

[Shamir79] Adi Shamir. How to share a secret. Communications of the ACM, 22 (11):612-613. ACM, November 1979.

[Siewiorek92] D. P. Siewiorek, R. S. Swarz “Reliable Computer Systems: design and evaluation”, Digital Press, Second Edition, 1992

[Soules03] Craig A. N. Soules, Garth R. Goodson, John D. Strunk, and Greg Ganger. Metadata efficiency in versioning file systems. Conference on File and Storage Technologies (San Francisco, CA, 31 March–02 April 2003), pages 43–57. USENIX Association, 2003.

[Spasojevic96] M. Spasojevic and M. Satyanarayanan. An empirical study of a wide-area distributed file system. ACM Transactions on Computer Systems, 14(2):200-222, May 1996.

[Stinson95] D. Stinson. Cryptography- Theory and Practice. CRC Press LLC, USA, 1995.

[Strunk 02] John D. Strunk, Garth R. Goodson, Adam G. Pennington, Craig A. N. Soules, and Gregory R. Ganger. Intrusion detection, diagnosis, and recovery with self-securing storage. Technical report CMU–CS–02–140. Carnegie Mellon University, 2002.

[Strunk00] John D. Strunk, Garth R. Goodson, Michael L. Scheinholtz, Craig A.N. Soules, and Gregory R. Ganger. Self-Securing Storage: Protecting Data in Compromised Systems. In the Proceedings of the 2000 OSDI Conference, October 2000.

[Tang92] D. Tang, R. K. Iyer “Analysis and Modeling of Correlated Failures in Multicomputer Systems”, IEEE Transactions on Computers, Vol. 41 No:5, May 1992

[Tichy80] W. F. Tichy. Software development control based on system structure description. PhD thesis. Carnegie-Mellon University, Pittsburgh, PA, January 1980.

[TPCC01] TPC BENCHMARK™ C Standard Specification, Revision 5.0, February 26, 2001, http://www.tpc.org/tpcc/spec/tpcc_current.pdf

[Tumblin01] Evelyn Tumlin Pierce. Self-adjusting quorum systems for byzantine fault tolerance. PhD thesis, published as Technical report CS–TR–01–07. Department of Computer Sciences, University of Texas at Austin, March 2001.

[Turner93] Roy M. Turner. “The Tragedy of the Commons and Distributed AI Systems.” Proceedings of the 12th International Workshop on Distributed Artificial Intelligence. 1993.

[Vesely81] W. E. Vesely, F. F. Goldber, N. H. Roberts, and D. F. Haasl. Fault Tree Handbook. U.S. Nuclear Regulatory Commission, NUREG-0492, Washington DC. 1981.

[Vo97] K. Vo, Y. Wang, et al. Xept: A Software Instrumentation Method for Exception Handling. The 8th International Symposium on Software Reliability Engineering, 1997.

[Vogels99] Werner Vogels. File system usage in Windows NT 4.0. ACM Symposium on Operating System Principles (Kiawah Island Resort, Charleston, South Carolina, 12-15 December 1999). Published as Operating System Review, 33(5):93-109. ACM, December 1999.

[Wong01] Theodore M. Wong, Jeannette M. Wing. Verifiable Secret Redistribution CMU SCS Technical Report CMU-CS-01-155, October 2001.

[Wong02a] Theodore M. Wong, Chenxi Wang, Jeannette M. Wing. Verifiable Secret Redistribution for Threshold Sharing Schemes. CMU SCS Technical Report CMU-CS-02-114, February 2002 (revised).

[Wong02b] T. Wong, C. Wang, J. Wing, Verifiable Secret Redistribution for Survivable Storage Systems. Computer Science Technical Report, School of Computer Science. 2002.

[Wong02c] T. M. Wong, C. Wang, and J. M. Wing. Verifiable Secret Redistribution for Archive Systems. Proceedings of the First International IEEE Security in Storage Workshop, December 2002.

[Wylie00] Jay J. Wylie, Michael W. Bigrigg, John D. Strunk, Gregory R. Ganger, Han Kiliccote, and Pradeep K. Khosla. Survivable information storage systems. IEEE Computer, 33(8):61-68, August 2000.

[Wylie01] Jay J. Wylie, Mehmet Bakkaloglu, Vijay Pandurangan, Michael W. Bigrigg, Semih Oguz, Ken Tew, Cory Williams, Gregory R. Ganger, and Pradeep K. Khosla. Selecting the right data distribution scheme for a survivable storage system. Technical report CMU–CS–01–120. CMU, May 2001.

[Wylie03a] Jay J. Wylie, Garth R. Goodson, Gregory R. Ganger, Michael K. Reiter. Safety and liveness proofs for a protocol family for versatile survivable storage infrastructures. Carnegie Mellon Parallel Data Lab Technical Report CMU-PDL-03-105. December 2003.

[Wylie03b] Jay J. Wylie, Garth R. Goodson, Gregory R. Ganger, Michael K. Reiter. Efficient Byzantine-tolerant erasure-coded storage. Carnegie Mellon Parallel Data Lab Technical Report CMU-PDL-03-104. December 2003.

[Ylonen96] Tatu Ylonen. SSH -- Secure login connections over the internet. USENIX Security Symposium (San Jose, CA). USENIX Association, 22-25 July 1996.

14 APPENDICES

14.1 Appendix A – PASIS Survivable Storage Systems

This appendix describes the technology characterization and survivability validation framework.

14.1.1 Technology Description & Survivability/Security Problem Addressed

The PASIS architecture combines decentralized storage system and data redundancy and encoding technologies to achieve survivable information storage [Wylie00]. Specifically, the security and availability policies of a PASIS system can survive failures and compromises of storage nodes, because information is entrusted to sets of nodes via well-chosen encoding and redundancy schemes. Some storage nodes can fail (or be eliminated by denial-of-service attacks), and the others will contain enough data to reconstruct stored information. Likewise, intruders (or insiders) can examine or manipulate the contents of some storage nodes, but they will not be able to compromise information confidentiality or integrity.

A PASIS system includes clients and servers. The servers, or storage nodes, provide persistent storage of shares; PASIS software on the clients provides all other aspects of PASIS functionality. Specifically, PASIS client agents communicate with collections of PASIS servers to collect necessary shares and combine them appropriately. To store information, PASIS client agents encode and partition it into shares, using some data distribution scheme, and store each share on a distinct storage node. The storage nodes verify each client request to ensure that only authorized users are allowed to read or write shares via the standard interface. Figure 1 illustrates the PASIS architecture.

Note that we have not specified a particular storage node protocol. In fact, the PASIS architecture can be used with any storage protocol, including NFS, AFS, CIFS, FTP, and HTTP. We are even exploring how mixes of protocols can be used in a single PASIS system.

Central to maximizing the benefits of survivable storage is the mindful selection of the *data distribution scheme* [Wylie01]. A data distribution scheme consists of a specific algorithm for data encoding & partitioning and a set of values for its parameters. There are many algorithms applicable to survivable storage,

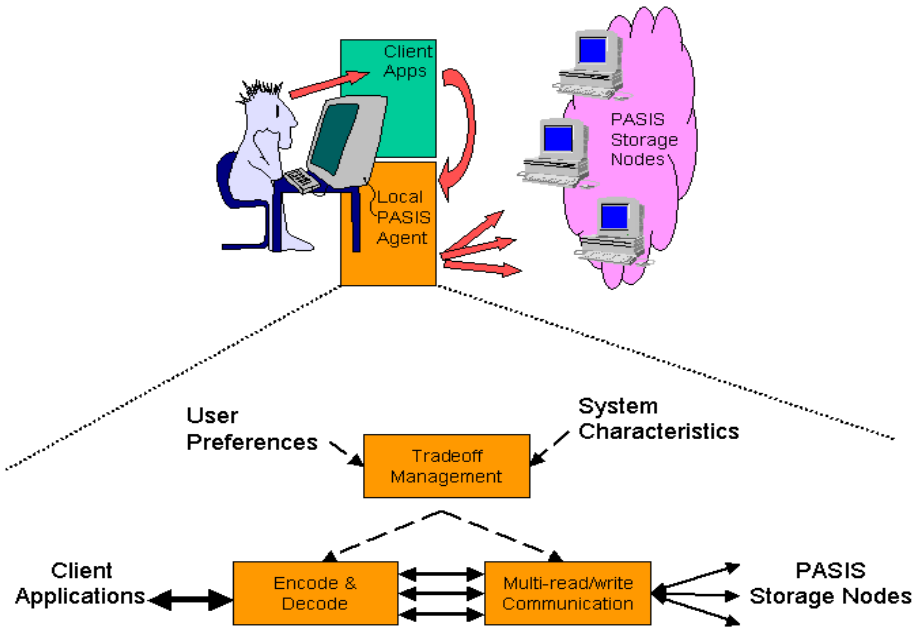


Fig A-1. PASIS agent software. Unmodified client applications interact with a PASIS storage system via a local agent, which encodes/decodes data and communicates with the many storage nodes. In the expanded view, data passes from left to right for writes, and from right to left for reads. The trade-off management box assists in system configuration, particularly in the choice of data distribution scheme.

including encryption, replication, striping, erasure-resilient coding, secret sharing, and various combinations. Each algorithm has one or more tunable parameters. The result is a large toolbox of possible schemes, each offering different levels of performance (throughput), availability (probability that data can be accessed), and security (effort required to compromise the confidentiality and integrity of stored data). For example, replication provides availability at a high cost in network bandwidth and storage space, whereas short secret sharing provides availability and security at lower storage and bandwidth cost but higher CPU utilization. Likewise, selecting the number of shares required to reconstruct a secret-shared value involves a trade-off between availability and confidentiality: if more machines must be compromised to steal the secret, then more must be operational to provide it legitimately.

We have made significant progress determining the performance cost of different data distribution schemes. Please see chapter 12 for more information on the database benchmarks used. The excerpt provides an example of our performance evaluation of data distribution schemes. We explore the trade-offs among performance, availability, and confidentiality in [Wylie01]. We have completed some further work on the confidentiality and integrity trade-offs in survivable storage, however the work is not yet complete. We examine the impact of correlated failures on the availability of survivable storage systems in [Bakkaloglu02a]. We have developed a mechanism for verifiable proactive repair within the PASIS architecture [Wong02c].

Because the PASIS architecture does not specify a single data distribution scheme, it is non-trivial to express its exact survivability features. However, there is sufficient commonality among systems conforming to this architecture for us to explain their basic features.

In addition to the PASIS architecture, we are exploring a server-side technology, called self-securing storage [Strunk00], wherein storage nodes internally version all data and audit all requests for a guaranteed amount of time, such as a week or a month. Self-securing storage can help security administrators to diagnose and recover from intrusions after the fact. Specifically, the node-maintained history information prevents intruders from destroying or undetectably tampering with stored information. Because there is no way for a storage node to differentiate successful intruders from legitimate users, it prevents anyone from doing these things. Our work on storage-based intrusion detection is presented in [Pennington02]. Current work is exploring how survivable storage architectures such as PASIS can take advantage of self-securing storage. Although self-securing storage helps storage systems to survive intrusions (in the diagnosis and recovery phases), the focus of this validation report is on PASIS exclusively.

14.1.2 Assumptions

Request authenticity and confidentiality assumptions

A1	Network eavesdropping is prevented.
A2	Forging of user credentials is prevented.
A3	Authenticated users are legitimate.

These assumptions, combined with the third mechanism below, serve to reduce the set of attacks handled to those that eliminate or compromise storage nodes.

Independence assumption

A4	The effort required to compromise multiple storage nodes scale at least somewhat with the number compromised (i.e., heterogeneous attacks are required).

This assumption is required to make data distribution useful against thoughtful attackers. Without it, data distribution will only help if the attackers employ random (or naïve) attack strategies.

Since there are many COTS implementations for each of many storage node interfaces (NFS, AFS, CIFS, etc.), storage node heterogeneity is achievable. As well, storage nodes can be distributed across distinct administrative domains. Storage nodes are also amenable to being physically distributed. At the scale of distributing storage nodes across a campus, no performance impact is expected. Except for errors in spe-

cific storage protocols that can be exploited by an attacker, achieving independent storage node failures appears to substantially be an issue of cost (equipment and people to administer the equipment).

Serial access assumption

A5	Only one client system accesses a particular datum at once.

This assumption is required for the base PASIS system described in Section 1. With the addition of inter-server communication (e.g., to employ quorum-based serialization [Malkhi00]), this assumption can be eliminated.

Implementation correctness assumptions

A6	System specifications match (or exceed) system requirements.
A7	PASIS libraries executing on true clients correctly distribute stored data.

These assumptions simply state that the software and configuration correctly perform the functions described in Section 1, configured to match the system's survivability requirements.

14.1.2.1 Residual risks and limitations

Beyond the assumptions, which must be dealt with via other mechanisms, the main residual security risk is that more storage node failures or compromises will occur than the chosen data distribution scheme (or, perhaps, any data distribution scheme) can accommodate. This risk will be particularly significant when multiple servers can be compromised rapidly because insufficient heterogeneity (see A4) exists in their implementations and configurations.

The PASIS architecture provides storage infrastructure survivability. It does not protect client systems or their ability to communicate with the storage system. For example, a denial-of-service attack on the client itself can prevent that client from accessing a PASIS storage system.

14.1.3 Attacks (or impairments)

The assumptions (particularly A1-A3) reduce the classes of attacks covered to those affecting storage nodes and client access to them. The effects of all such attacks fall into three broad categories:

Storage node elimination attacks

V1	One or more storage nodes may simply fail (for whatever reason).
V2	One or more storage nodes may become inaccessible because of a network failure or partition (for whatever reason).
V3	One or more storage nodes may be "taken out" by a successful denial-of-service attack (for whatever vulnerability it targets).

These attacks all have the effect of making a storage node unavailable to clients, thereby denying access to its contents.

Storage confidentiality attacks

V4	An intruder who compromises one or more storage nodes may examine the stored contents of those nodes (individually or collectively).
V5	Malicious system administrators or other insiders may examine the stored contents of one or more storage nodes (individually or collectively).

These attacks all have the effect of allowing an intruder to look at the data stored on a particular storage node. (Note: many data distribution schemes generate non-cleartext data shares.)

Storage integrity attacks

V6	An intruder who compromises one or more storage nodes may modify the stored contents of those nodes (individually or collectively).
V7	Malicious system administrators or other insiders may modify the stored contents of one or more storage nodes (individually or collectively).

These attacks all have the effect of allowing an intruder to manipulate the data stored on a particular storage node (temporarily or permanently).

14.1.4 Survivability and Security Attributes (Goals)

The PASIS architecture is designed to protect the confidentiality (C), integrity (I), and availability (AV) of stored data from failures within and attacks on the storage infrastructure. PASIS does not directly address the attributes of authentication (AU) and non-repudiation (NR). (Note: self-securing storage can assist security administrators with diagnosis of and recovery from user and client authentication failures.)

14.1.5 Comparison with Other Systems

The PASIS architecture generalizes the survivable storage architectures of a number of previous and current research projects. By doing so, we are able to explore the trade-offs associated with different system configurations. In particular, the performance, availability, and confidentiality of different data distribution schemes are evaluated.

14.1.6 Survivability and Security Mechanisms

Although there are many aspects to building a complete PASIS system, three main mechanisms suffice to explain how the survivability goals are satisfied. Two are really attributes of the same mechanism (use of a well-chosen data distribution scheme).

Data Redundancy Mechanism

M1	Data are encoded and distributed amongst storage nodes such that subsets of storage nodes can collectively provide it upon request.

The description of this mechanism is purposely vague, because there are various ways that this can be achieved, including various forms of replication, threshold encoding, erasure correction coding, and erasure correction coding. For threshold schemes, any *m* of the *n* shares are sufficient. For other schemes, specific shares (stored on specific storage nodes) may be required.

Data Encoding Mechanism

M2	Data are encoded and distributed amongst storage nodes such that small subsets of storage nodes cannot reveal it to anyone (including an intruder or administrator).

The description of this mechanism is purposely vague, because there are various ways that this can be achieved, including various forms of encryption, secret sharing, and other information dispersal algorithms. For secret sharing schemes, *m* of the *n* shares must be obtained to discover the cleartext information. For encryption schemes, both the ciphertext and the encryption key (which should be stored separately) must be obtained.

Access Control Enforcement Mechanism

M3	Each server only performs authorized reads/writes. That is, servers enforce the access control policies corresponding to each piece of information.

Combined with the assumption that user authentication works correctly, this mechanism prevents unauthorized users from reading or writing information via the normal interface. In the most appropriate system configuration, user authentication is performed by an entity other than the client OS (e.g., a Kerberos

server or the storage nodes themselves), such that compromising a client OS does not compromise the authentication mechanism.

14.1.7 Rationale

14.1.7.1 Goal vs. Impairment Matrix

<table>
<tr><th>Operation</th><th>AV</th><th>I</th><th>C</th><th>AU</th><th>NR</th></tr>
<tr><td>V1</td><td rowspan="3">1: M1</td><td rowspan="3">N/A</td><td rowspan="3">N/A</td><td rowspan="5">N/A</td><td rowspan="5">N/A</td></tr>
<tr><td>V2</td></tr>
<tr><td>V3</td></tr>
<tr><td>V4</td><td rowspan="2">N/A</td><td rowspan="2">N/A</td><td rowspan="2">2: M2</td></tr>
<tr><td>V5</td></tr>
<tr><td>V6</td><td rowspan="2">N/A</td><td rowspan="2">3: M1, M2</td><td rowspan="2">N/A</td></tr>
<tr><td>V7</td></tr>
</table>

14.1.7.2 Claims

1. Redundant information {M1} stored on other storage nodes allows the system to compensate for the loss of a subset. This is a well-known, proven attribute of the data distribution schemes under consideration [Berlekamp84, Blahut83, Patterson88, Rabin89].

Under assumption A4 this is a *class A* claim, i.e., basic fault tolerance analysis shows that as the degree of data redundancy increases, the availability increases.

Under a weakening of assumption A4, specifically that some benign faults are correlated (e.g., some subset of storage nodes share a power source) this is a *class B* claim. In [Bakkaloglu02] we examine web server availability data and campus area desktop computer availability data. We further demonstrate that correlated failures can be detected over time. Given the ability to identify correlated availability, we can employ an intelligent server selection strategy so that storage nodes used in a PASIS-like scheme would not exhibit correlated failures [Bakkaloglu021].

2. Because data from multiple storage nodes {M2} are required to reconstruct the information (under most schemes being considered), examining the contents of a subset can be made to yield no information. This is a well-known, proven attribute of the data distribution schemes under consideration [Menezes97, Schneier96, Shamir79, Stinson95].

Again, under assumption A4, this is a *class A* claim. Data distribution schemes such as secret sharing and short secret sharing can tolerate up to *m* shares being observed without compromising the confidentiality of the stored data. This claim is considered in the context of V4 and V5.

Under V4 (intruder attacks on storage nodes), the confidentiality of the system scales with the number of distinct storage nodes an intruder needs to compromise to collect *m* shares. If all *n* storage nodes that host shares are not distinct from the perspective of an attacker, then the number of storage nodes an intruder needs to compromise is less than *m* (i.e., a single attack can compromise more than one storage node).

Under V5 (malicious system administrators), an inside attack by the storage administrator must involve *m* storage administrators. Given data distributed across administrative domains, *m* administrators must now collaborate to compromise the data confidentiality.

3. Because data from multiple storage nodes {M2} are required to reconstruct the information (under most schemes being considered), modifying the contents of a subset can be made to affect no information. The redundant information {M1} from other storage nodes can be used to detect and correct the errors (i.e., modifications). This is a well-known, proven attribute of the data distribution schemes under consideration [Menezes97, Schneier96, Stinson95].

This claim is almost identical in form to claim 2. The only distinction is that in the case of V4, intruders need to formulate an attack that enables them to modify stored shares. An attack that enables an intruder to observe a share is sufficient for a confidentiality attack, but insufficient for an integrity attack. Otherwise, the integrity claim is the same as the confidentiality claim.

14.1.7.3 Verification Techniques

The survivability features of the PASIS architecture come from its use of well-chosen data distribution schemes. The availability, confidentiality, and integrity attributes of these schemes are both well-known and proven, so further verification should focus on two aspects: correct implementation & configuration and the independence assumption. The former can rely on standard (and emerging) software assurance and testing techniques. The latter is an open research question that is difficult to verify convincingly. However, as mentioned earlier, protocol, implementation, and geographic diversity of storage nodes is achievable today with sufficient funds.

Our project continues to explore the independence assumption (see [Bakkaloglu02] for our work on correlated benign faults) and how to make it more true (e.g., by allowing different server implementations and even different protocols in the same PASIS system).

14.1.8 Cost/Benefit Analysis

14.1.8.1 Cost Considerations

Development. Development of the PASIS libraries and agent software for relevant client platforms is a one-time cost.

Storage node count. Acquisition cost above that of a functionally equivalent site is due mainly to the increased number of storage nodes (over which encoded data is distributed).

System administration. Beyond the cost of administering a larger number of storage nodes, achieving the heterogeneity required to make survivable storage possible is likely to require use of storage nodes from multiple different vendors (e.g., a Sun NFS server, a Network Appliances filer, a Snap server, and a Linux NFS server). Retaining system administrators capable of managing all of these system types will increase cost, and may not even be possible in some circumstances. Further, it may be appropriate to have different people managing each subset of storage nodes, again increasing cost.

Performance. Perhaps the largest concern with survivable storage is the performance cost of encoding/decoding data and communicating with multiple storage nodes for each user request. Quantifying and mitigating this cost is a central focus of our project. Our first analysis of the performance cost of different data distribution schemes is presented in [Wylie01]. More recently, we analyzed the performance cost of survivable storage in the context of the Joint Battlespace Infosphere (JBI) project. Specifically, we evaluated the performance impact of using PASIS as the back end store for an Oracle database running TPC-H. TPC-H was determined to be comparable to the expected workload in the JBI project. A chapter from the final report on the JBI-PASIS evaluation is included as Appendix A of this validation report.

14.1.8.2 Benefits

Systems conforming to the PASIS architecture can survive failures and compromises of parts of the storage infrastructure, as described throughout this technology characterization.

14.2 Appendix B – Server Lists

Appendix B-1: List of the Web Servers used in the experiments in Chapter 4.

www.cmu.edu	www.moscowtimes.ru	www.microsoft.com
www.cs.cmu.edu	www.ukindex.co.uk	www.ibm.com
www.ece.cmu.edu	www.ukdirectory.co.uk	www.panasas.com
www.berkeley.edu	www.mod.uk	www.sun.com
www.mit.edu	www.state.mn.us	www.hp.com
www.harvard.edu	www.state.hi.us	www.ananzi.co.za
www.cs.ucsd.edu	www.cao.go.jp	www.southafrica.co.za
www.utexas.edu	www.jda.go.jp	www.csu.edu.au
www.florida.edu	www.beijing.gov.cn	www.made-in-denmark.dk
www.nyu.edu	www.gio.gov.tw	www.snb.ch
www.gwu.edu	www.gov.za	www.ual.com
www.upr.clu.edu	pmindia.nic.in	www.aa.com
www.metu.edu.tr	www.indianembassy.org	www.wtca.org
www.encyclopedia.com	www.mfa.gov.tr	www.afghanistanfoundation.org
www.jeremyshaffer.com	www.turkey.org	www.onefootball.com
www.freebsd.org	armedforces.nic.in	www.unesco.org
www.linux.org	www.pakistan-embassy.com	www.afghan-web.com
www.gnu.org	www.sc.gov.jm	vlib.org
www.asiasource.org	www.cabinet.gov.jm	www.dictionary.com
www.newyorktimes.com	www.cuba.cu	www.bl.uk
www.canoe.ca	www.austrade.gov.au	www.literature.org
www.bbc.co.uk	www.admin.ch	www.druglibrary.org
www.japantimes.co.jp	www.gov.ru	www.oclc.org
www.nasda.go.jp	www.firstgov.gov	www.galaxy.com
www.sudan.net	www.whitehouse.gov	www.usenix.org
www.milliyet.com.tr	usembassy.state.gov	msdn.microsoft.com
www.pca.gov.sa	president.gov.al	www.acm.org
www.mg.co.za	www.gov.mk	www.online-library.org
www.afrika.no	www.presidency.gov.eg	www.library.cmu.edu
www.jpost.com	www.loc.gov	digital.library.upenn.edu
www.suntimes.co.za	www.nasa.gov	www.sba.gov
www.pressnet.or.jp	www.emc.com	www.amnesty.org
www.heraldsun.news.com.au	www.intel.com	www.prodigy.com

Appendix B-2: Server Correlations

The following table contains the r-values calculated as part of the inter-server performance correlation analysis (Section 6.3.1). All server pairs with r-values of more than 0.2 are listed here.

Server 1	Server 2	r- value
asia.cnn.com	www.cnn.com	0.557655
armedforces.nic.in	pmindia.nic.in	0.40282
msdn.microsoft.com	www.berkeley.edu	0.400202
www.cs.ucsd.edu	www.ece.cmu.edu	0.3659
www.ece.cmu.edu	www.microsoft.com	0.336335
www.cs.ucsd.edu	www.harvard.edu	0.331532
www.berkeley.edu	www.cabinet.gov.jm	0.320242
www.ece.cmu.edu	www.library.cmu.edu	0.304477
www.ece.cmu.edu	www.harvard.edu	0.302797
msdn.microsoft.com	www.cabinet.gov.jm	0.297736
msdn.microsoft.com	www.microsoft.com	0.283816
www.berkeley.edu	www.fifa.com	0.279116
www.cs.ucsd.edu	www.library.cmu.edu	0.272334
msdn.microsoft.com	www.weather.com	0.271211
www.ananzi.co.za	www.suntimes.co.za	0.269138
www.cs.ucsd.edu	www.microsoft.com	0.267225
www.berkeley.edu	www.cs.cmu.edu	0.262694
www.berkeley.edu	www.weather.com	0.261328
www.cao.go.jp	www.japantimes.co.jp	0.247262
www.cabinet.gov.jm	www.weather.com	0.245397
msdn.microsoft.com	www.fifa.com	0.23886
www.gov.ru	www.gov.za	0.234039
www.cmu.edu	www.ece.cmu.edu	0.233815
www.gov.za	www.mg.co.za	0.232518
www.cabinet.gov.jm	www.fifa.com	0.230866
games.yahoo.com	www.freebsd.org	0.227876
www.berkeley.edu	www.microsoft.com	0.224917
usembassy.state.gov	www.microsoft.com	0.220802
www.berkeley.edu	www.metu.edu.tr	0.220327
www.berkeley.edu	www.ece.cmu.edu	0.218202
www.library.cmu.edu	www.microsoft.com	0.216106
www.ece.cmu.edu	www.hp.com	0.215056
www.amnesty.org	www.cabinet.gov.jm	0.215012
msdn.microsoft.com	usembassy.state.gov	0.213936
www.fifa.com	www.weather.com	0.213245
www.berkeley.edu	www.ual.com	0.213194
www.cs.ucsd.edu	www.firstgov.gov	0.211438
usembassy.state.gov	www.ece.cmu.edu	0.210704
www.berkeley.edu	www.pak.gov.pk	0.209283
www.cmu.edu	www.cs.ucsd.edu	0.208952
www.austrade.gov.au	www.cabinet.gov.jm	0.207661
www.ece.cmu.edu	www.firstgov.gov	0.207587
www.hp.com	www.microsoft.com	0.207396
armedforces.nic.in	ww.gov.za	0.2066
armedforces.nic.in	www.pak.gov.pk	0.204578
www.harvard.edu	www.microsoft.com	0.204135
msdn.microsoft.com	www.ece.cmu.edu	0.202033
www.berkeley.edu	www.washingtonpost.com	0.20155
www.cabinet.gov.jm	www.washingtonpost.com	0.201471

Appendix B-3. All servers, and corresponding server numbers. Some numbers are missing because their servers never responded correctly to the automated queries.

Server number	URL
	www.cuba.cu
4	www.upr.clu.edu
5	www.harvard.edu
6	www.made-in-denmark.dk
7	www.moscowtimes.ru
8	www.library.cmu.edu
9	www.florida.edu
10	www.ananzi.co.za
11	www.sba.gov
12	www.utexas.edu
13	www.dictionary.com
14	www.europa.eu.int
15	www.afghan-web.com
16	www.bl.uk
17	msdn.microsoft.com
18	www.jpost.com
19	www.asiasource.org
20	www.ukdirectory.co.uk
21	www.usenix.org
22	www.cao.go.jp
23	www.ukindex.co.uk
24	www.fifa.com
25	www.milliyet.com.tr
26	www.panasas.com
27	www.emc.com
28	www.firstgov.gov
29	www.online-library.org
30	www.brasil.gov.br
31	www.ericsson.com
32	www.jeremyshaffer.com
33	www.loc.gov
35	www.afrika.no
36	www.washingtonpost.com
37	www.nasda.go.jp
38	www.freebsd.org
40	www.aa.com
41	www.redcross.org
42	www.mit.edu
43	yahoo.co.jp
44	www.mg.co.za
45	www.suntimes.co.za
46	www.sun.com

Server number	URL
47	www.canoe.ca
48	www.ual.com
49	www.heraldsun.news.com.au
50	vlib.org
51	www.gov.mk
52	www.gov.ru
53	www.gov.za
54	www.hp.com
55	www.literature.org
56	www.turkey.org
58	pmindia.nic.in
60	www.cs.ucsd.edu
61	www.sudan.net
62	www.acm.org
63	www.mod.uk
64	www.presidency.gov.eg
65	asia.cnn.com
66	www.meninblack.com
67	www.gnu.org
68	games.yahoo.com
69	www.state.hi.us
70	armedforces.nic.in
71	www.state.mn.us
72	www.defenselink.mil
73	www.encyclopedia.com
74	www.snb.ch
75	www.ece.cmu.edu
76	www.admin.ch
77	president.gov.al
78	usembassy.state.gov
79	www.deccanherald.com
80	www.metu.edu.tr
81	www.gwu.edu
82	www.ibm.com
83	www.nyu.edu
84	www.druglibrary.org
85	www.berkeley.edu
86	www.state.gov
87	www.pakistan-embassy.com
88	digital.library.upenn.edu
89	www.beijing.gov.cn
90	www.indianembassy.org

Server number	URL
91	www.oclc.org
92	www.galaxy.com
93	www.intel.com
96	www.austrade.gov.au
97	www.weather.com
98	www.sc.gov.jm
99	www.microsoft.com
100	www.csu.edu.au
101	mail.yahoo.com
102	www.mfa.gov.tr
103	www.linux.org
104	www.unesco.org
105	www.amnesty.org
106	www.pressnet.or.jp
107	www.jda.go.jp
108	www.southafrica.co.za
110	www.cnn.com
111	www.cabinet.gov.jm
112	www.whitehouse.gov
113	www.japantimes.co.jp
114	www.onefootball.com
116	www.pca.gov.sa